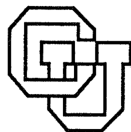
A Second Simulation of the Gries/Dijkstra Design Process

Robert B. Terwilliger

CU-CS-618-92 October 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A Second Simulation 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 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 our second simulation of the Gries/Dijkstra design process. Specifically, we describe a number of new cliches, verify their correctness, and give an example of their use in a design derivation for Kemmerer's Library Problem.

1. Introduction

The development of software consumes a significant portion of our society's economic resources [7]. 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 [11]; 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* [6, 10, 22, 27, 40, 42]. These techniques require that a software item or the process used to produce it be examined and evaluated by a group of knowledgeable personnel. Even more information can be gathered through *process programming* [14-16, 26, 32, 33]; in other words, when software processes are described 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. To a certain extent, process programming overlaps with previous work on knowledge-based software engineering [1, 8, 9, 12, 17, 19, 20, 28-31]. An important aspect of many of these projects is 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 [36-39]. 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, 18, 23-25]. 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 [4, 5, 13]. 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 one iteration of our three step procedure on the Gries/Dijkstra design process [36-39], and we have performed steps one (walkthrough) and two (simulation) for a second time. Our simulation programs are 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.

In this paper, we describe our second simulation of the Gries/Dijkstra design process. The architecture

of the current system is identical to that used in the first iteration [37,38]; however, the two differ in that the second simulation uses a number of new cliches that we feel better reflect the process described in [13]. Specifically, an attempt is made to formally describe the "replace a constant with a variable" method for constructing an invariant from a post-condition, and the "decrease then restore" strategy for separating concerns in the construction of a loop body.

In the remainder of this paper, we describe our second simulation in more detail. In section two we give some background on the Gries/Dijkstra design process, and in section three we review our simulation architecture. In section four we present the new cliches and verify their correctness, and in section five we describe how they can be used to generate a design for a solution to Kemmerer's Library Problem. 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 [4,5,13]; our view of the method is based primarily on [13]. 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 is 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.

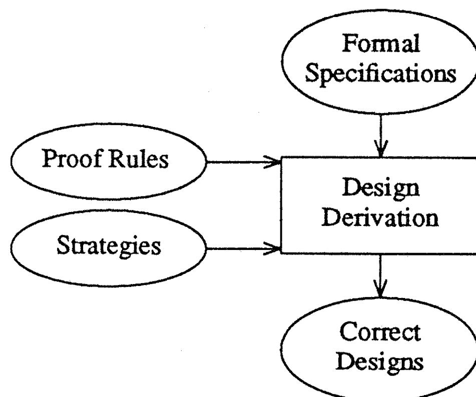


Figure 1. Gries/Dijkstra Design Process

For example, Kemmerer's Library problem has received considerable attention in the software engineering literature and has been formally specified a number of times [21,41,43]. 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 [36] 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 [13]. 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 the loop, we have already proven the design correct. The rest of this example is presented in [36].

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 simulations.

3. Simulation Architecture

Figure 3 shows a pictorial representation of the design process implemented in our simulations. 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

```
{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

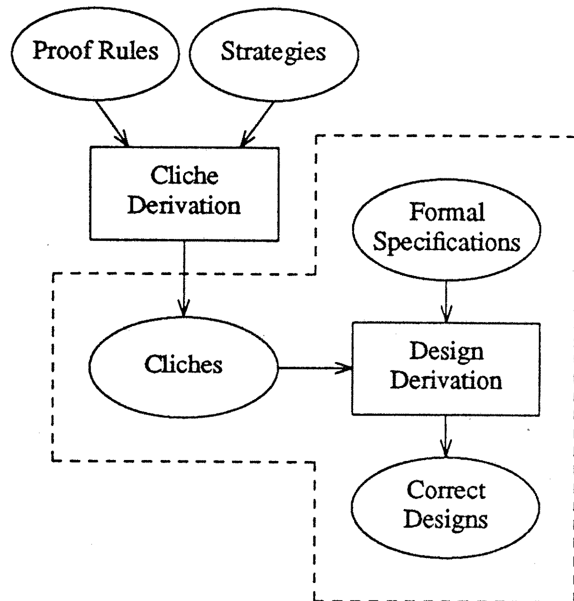


Figure 3. Design Process as Simulated

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 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.

We have constructed a running system based on the architecture in Figure 3. The program was designed using guarded commands and its correctness rigorously verified [37]. 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 [34,35]. The implementation is somewhat sketchy, especially the logic manipulation and theorem proving routines; however, it does demonstrate that the design is fundamentally correct.

Our simulation engine 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 five. Two of these were developed for the first simulation [37,38]. The "simple_assignment" cliche generates (multiple) assignment statements, and the "simple_if_then_else" cliche generates two branch if-then-else statements.

Three new cliches will be developed here. 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 applies to situations where the body of a loop must conditionally apply an operation to the iteration element.

Figure 4 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 ∧ E1 ∨ B2 ∧ E2" if: "B1" is the logical negation of "B2"; "S1" is correct with respect to pre- and post-conditions "Q ∧ B1" and "B1 ∧ E1" respectively; and "S2" is correct with respect to "Q ∧ B2" and "B2 ∧ E2".

Figure 5 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 with respect to a pre-condition "Q" and post-condition "R" if "S0" and "S1" satisfy their specifications and three additional conditions are met.

First, "C" must a constant in the post-condition. This is represented as membership in a set-valued function on the appropriate formula. Second, the tuple "(C,E,V,Rng)" must be an element of the relation "mkvariable", where "V" is a variable, "E" is an expression in "C" and "V", and "Rng" is a boolean expression restricting the range of "V". Third, the loop invariant must be equivalent to "Rng" conjoined with the result of substituting "E" for "C" in the post-condition.

For this cliche to be correct, every tuple of the mkvariable relation must satisfy a certain property. Specifically, the restrictions on "V" ("Rng"), combined with the fact that "E" is not equal to "C" must imply that the difference between the size of "C" and the size of "E" is greater than or equal to zero.

```
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 4. Simple Cliches

```
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 5. *Simple_Replace_Constant* Cliche

Property 1: (C,E,V,Rng) $\in$ mkvariable =>
$Rng \wedge E \neq C \Rightarrow (|C| - |E|) \geq 0$

We can now argue for the correctness of "simple_replace_constant" using the proof rules given in [13].

Theorem 1: {Q} simple_replace_constant {R}

{Q} S0 {P} DO $\{P \wedge \neg B\}$ {R}
where P: $Rng \wedge (R)^{C}_{E}$, B: $E \neq C$

1) {Q} S0 {P} by assumption
2) {P} DO $\{P \wedge \neg B\}$ by lemma 1
3) $Rng \wedge (R)^{C}_{E} \wedge E=C \Rightarrow R$

therefore, {Q} simple_replace_constant {R}.

Lemma 1: {P} DO $\{P \wedge \neg B\}$
where P: $Rng \wedge (R)^{C}_{E}$, B: $E \neq C$, t: $|C| - |E|$,
Q1: $P \wedge B \wedge t=t1$, R1: $P \wedge t<t1$

1) $\{P \wedge B\}$ {Q1} S1 {R1} {P} by lemma 1.1
2) $Rng \wedge (R)^{C}_{E} \wedge E \neq C \Rightarrow (|C| - |E|) \geq 0$
$Rng \wedge E \neq C \Rightarrow (|C| - |E|) \geq 0$ by property 1
3) $\{P \wedge B\}$ t1:=t {Q1} S1 {R1} {t<t1}
by lemma 1.2

therefore {P} DO $\{P \wedge \neg B\}$.

Lemma 1.1: $\{P \wedge B\}$ {Q1} S1 {R1} {P}
where Q1: $P \wedge B \wedge t=t1$, R1: $P \wedge t<t1$

1) $P \wedge B \Rightarrow P \wedge B \wedge t=t1$ defines t1
2) {Q1} S1 {R1} by assumption
3) $P \wedge t<t1 \Rightarrow P$

therefore $\{P \wedge B\}$ {Q1} S1 {R1} {P}.

Lemma 1.2: $\{P \wedge B\}$ t1:=t {Q1} S1 {R1} {t<t1}
where Q1: $P \wedge B \wedge t=t1$, R1: $P \wedge t<t1$

1) $\{P \wedge B\}$ t1:=t $\{P \wedge B \wedge t=t1\}$
2) {Q1} S1 {R1} by assumption
3) $P \wedge t<t1 \Rightarrow t<t1$

therefore $\{P \wedge B\}$ t1:=t {Q1} S1 {R1} {t<t1}.

For the purpose of this paper, the proof of "simple_replace_constant" is now complete; however, this is only the first of three cliches we must present and verify.

Continuing, Figure 6 shows 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. Specifically, the cliche states that the two statement sequence "S0 ; S1" is correct with respect to precondition "Q" and post-condition "R" if: "S1" satisfies its specification; the tuple "(T,V,E,S0,F,D)" is an element of the relation "decrease_bnd"; and "S1" does not modify any of the variables referenced in "T".

Here, "T" is an integer function (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 (reflecting

```
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 6. *Decrease_Then_Restore* Cliche

additional facts concerning the modification), and "D" is a declaration (of the iteration variable).

For this cliche to be correct, every tuple of the "decrease_bnd" relation must satisfy three properties. "S0" must decrease "T"; "F" must be true after "S0" completes; and for any formula "W", if "W" is true before "S0" executes, then "W" with "E" substituted for "V" must be true after "S0" completes.

Property 2: (T,V,E,S0,F,D) $\in$ decrease_bnd =>

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

We can now argue for the correctness of "decrease_then_restore" using the proof rules given in [13]. However, to assist in this endeavor we will first present a fully annotated version of the cliche body.

```
{Q: P ∧ B ∧ T=t1}
< S0 >
{Q': P[[ V / E ]] ∧ F ∧ T=t2 ∧ T<t1}
{Q1: P[[ V / E ]] ∧ F}
< S1 >
{R1: P}
{R': P ∧ T=t2 ∧ T<t1}
{R: P ∧ T<t1}
```

Theorem 2: {Q} decrease_then_restore {R}

{Q} S0 {Q'} {Q1} S1 {R1} {R'} {R}
where R': $P \wedge T=t2 \wedge T<t1$, R: $P \wedge T<t1$

1) {Q} S0 {Q'} by lemma 2
2) {Q'} {Q1} S1 {R1} {R'} by lemma 3
3) $P \wedge T=t2 \wedge T<t1 \Rightarrow P \wedge T<t1$

therefore, $\{Q\}$ decrease_then_restore $\{R\}$.

Lemma 2: $\{Q\}$ S0 $\{Q'\}$

where $Q: P \land B \land T{=}t1$,
$Q': (P)^V_B \land F \land T{=}t2 \land T{<}t1$

1) $\{P \land B \land T{=}t1\}$ S0 $\{(P)^V_B\}$ by property 2.3
2) $\{P \land B \land T{=}t1\}$ S0 $\{F\}$ by property 2.2
3) $\{P \land B \land T{=}t1\}$ S0 $\{T{=}t2\}$ defines t2
4) $\{P \land B \land T{=}t1\}$ S0 $\{T{<}t1\}$ by property 2.1

therefore, $\{Q\}$ S0 $\{Q'\}$.

Lemma 3: $\{Q'\}$ $\{Q1\}$ S1 $\{R1\}$ $\{R'\}$

where $Q': (P)^V_B \land F \land T{=}t2 \land T{<}t1$,
$R': P \land T{=}t2 \land T{<}t1$,
$Q1: (P)^V_B \land F$, $R1: P$

1) $(P)^V_B \land F \land T{=}t2 \land T{<}t1 \Rightarrow (P)^V_B \land F$
2) $\{(P)^V_B \land F\}$ S1 $\{P\}$ by assumption
3) $\{T{=}t2 \land T{<}t1\}$ S1 $\{T{=}t2 \land T{<}t1\}$
because $modify(S1) \cap use(T) = \varnothing$

therefore, $\{Q'\}$ $\{Q1\}$ S1 $\{R1\}$ $\{R'\}$.

For the purposes of this paper, we will now consider the proof of "decrease_then_restore" complete and proceed to the third new cliche developed for this simulation.

Figure 7 show the "conditional_operation" cliche. This cliche applies to situations where the body of a loop must conditionally apply an operation to the iteration element. 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. Each iteration modifies the result depending on whether the item satisfies a certain property.

```
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 6. *Conditional_Operation* Cliche

The post-condition of the cliche states that the local set ("Lset") is a subset of the original ("Set"), and that the result variable ("Var") is equal to the value of "Iop(Set-Lset,Cond)". In other words, that "Var" is equal to the value of an iteration operator applied to the difference of the original and local sets and a certain condition. In the cliche, "Lset" is a set containing all the items still to be considered, while "Lvar" is the item currently being processed. The loop iterates over all the items in "Set", and if the item in question satisfies "Cond" then "Var" is set to "Op(Var,Lvar)".

For this cliche to be correct, every tuple "(Iop(Set,Cond),Op(Var,Lvar))" in "iteration_ops" must satisfy two properties. For each element being considered, if the condition holds then the new result can be computed from the old by applying the given operator. On the other hand, if the condition does not hold then the new result is identical to the old.

Property 3: $(Iop(Set,Cond),Op(Var,Lvar)) \in iteration_ops \Rightarrow$

1) $(Lvar \in Set \land Cond(Lvar) \Rightarrow Iop(Set,Cond) = Op(Iop(Set-Lvar,Cond),Lvar))$
2) $(Lvar \in Set \land \neg Cond(Lvar) \Rightarrow Iop(Set,Cond) = Iop(Set-Lvar,Cond))$

We can now argue for the correctness of "conditional_operation" using the standard proof rules.

Theorem 3: $\{Q\}$ conditional_operation $\{R\}$

$\{Q\}$ $\{Q1\}$ S1 $\{R1\}$ $\{R\}$

1) $Q \Rightarrow Var{=}VAR$ defines VAR
2) $\{Q1\}$ S1 $\{R1\}$ by assumption
3) $Q \land Var{=}VAR \Rightarrow (Q)^{Var}_{VAR}$
3) $\{(Q)^{Var}_{VAR}\}$ S1 $\{(Q)^{Var}_{VAR}\}$
because $modify(S1) \cap use((Q)^{Var}_{VAR}) = \varnothing$
3) $(Q)^{Var}_{VAR} \land R1 \Rightarrow R$ by Lemma 4

therefore, $\{Q\}$ conditional_operation $\{R\}$.

Lemma 4: $(Q)^{Var}_{VAR} \land R1 \Rightarrow R$

where $(Q)^{Var}_{VAR}$: $LSET \subseteq Set \land VAR = Iop(Set\text{-}LSET,Cond) \land Lset{=}LSET\text{-}Lvar \land Lvar \in LSET$
$R1: \neg Cond(Lvar) \land Var{=}VAR \lor Cond(Lvar) \land Var{=}Op(VAR,Lvar)$
$R: Lset \subseteq Set \land Var = Iop(Set\text{-}Lset,Cond)$

1) $(Q)^{Var}_{VAR} \Rightarrow LSET \subseteq Set \land$
$T1: (Lset{=}LSET\text{-}Lvar \land Lvar \in LSET)$
$\Rightarrow Lset \subseteq Set$
2) $(Q)^{Var}_{VAR} \Rightarrow T2: (VAR = Iop(Set\text{-}LSET,Cond))$
3) $R1 \Rightarrow T3: (\neg Cond(Lvar) \land Var{=}VAR) \lor$

T4: (Cond(Lvar) ∧ Var=Op(VAR,Lvar))
4) T1 ∧ T2 ∧ T3 => Var = Iop(Set-Lset,Cond)
by property 3.2
5) T1 ∧ T2 ∧ T4 => Var = Iop(Set-Lset,Cond)
by property 3.1
6) Lset⊆Set ∧ Var=Iop(Set-Lset,Cond) => R
therefore, $(Q)^{Var}_{VAR} \wedge R1 \Rightarrow R$

For the purpose of this paper, we will now consider the proof of "conditional_operation" complete, and with it our descriptions and verifications of the new cliches developed for our second simulation. Strictly speaking, we have not "proven" the cliches correct; however, the arguments presented are rigorous, and significantly increase our belief in the validity of these constructs.

Although the cliches presented in this section are fairly simple, they are still much more complex than the engine which applies them to create designs from specifications. 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 our simulation system 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 "simple_replace_constant" 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}
< simple_replace_constant >
{R}
```

The system selects the constant "users" from the postcondition, and the relation "mkvariable" produces the following tuple.

```
C   = users
E   = users-usrs
V   = usrs:set(users)
Rng = usrs⊆users
```

Notice that for the present purpose there is no distinction between a non-modifiable variable and a constant. Substituting "users-usrs" for "usrs" in the post condition and conjoining "usrs⊆users" we obtain the following loop invariant.

```
P:  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 8.

Application of the "simple_assignment" cliche to "S0" produces the following multiple assignment to initialize the loop.

```
who_has,usrs := {},users
```

We can see that the "decrease_then_restore" cliche is applicable to the specification of the loop body.

```
{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 8. Instantiated *Simple_Replace_Constant* Cliche

```
{Q: P ∧ B ∧ T=t1}
< decrease_then _restore >
{R: P ∧ T<t1}
```

The specification and cliche unify as follows.

```
P    = usrs⊆users ∧
       who_has =
            {u∈users-usrs:
                       corec(u,b)∈checks}
B    = usrs≠{}
T    = |usrs|
```

And the relation "decrease_bnd" produces the following tuple.

```
T    = |usrs|
V    = usrs
E    = USRS
S0   = choose(usrs,usr); usrs:=usrs-usr
F    = usrs=USRS-usr ∧ usr∈USRS
D    = usr:user
```

Therefore, instantiation of the "decrease_then_restore" cliche 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 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.

We can see that the "conditional_operation" cliche is applicable to the remaining specification.

```
{Q: LSET⊆Set ∧
    Var = Iop(Set-LSET,Cond) ∧
    Lset=LSET-Lvar ∧ Lvar∈LSET}
< conditional_operation >
{R: Lset⊆Set ∧
    Var = Iop(Set-Lset,Cond)}
```

The specification and cliche unify as follows.

```
Lset    = usrs
LSET    = USRS
Lvar    = usr
Set     = users
Var     = who_has
Cond    = corec(u,b)∈checks
Iop     = "set_of_all"
```

Therefore, instantiation of the "conditional_operation" cliche produces the following body for the loop.

```
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 loop body must still be completed before the design is finished. We can see that the "simple_if_then_else" cliche is applicable to the specification.

```
{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 →
    {Q1:who_has=WHO_HAS ∧
            corec(usr,b)∈checks}
    < S1 >(who_has:inout set(user));
    {R1: corec(usr,b)∈checks ∧
            who_has=WHO_HAS+usr}
 [] 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}
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 [4,5,13]. 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 one iteration of our three step procedure on the Gries/Dijkstra design process [36-39], and we have performed steps one (walkthrough) and two (simulation) for a second time. 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 method 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 system based on our two level architecture. The program was designed using guarded commands and its correctness rigorously verified [37]. 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.

In this paper, we have described our second simulation of the Gries/Dijkstra design process. The architecture of the current system is identical to that used in the first simulation [37,38]; however, the two differ in that the second simulation uses a number of new cliches that we feel better reflect the process described in [13]. Specifically, an attempt was made (with some success) to formally describe the "replace a constant with a variable" method for constructing an invariant from a post-condition, and the "decrease then restore" strategy for separating concerns in the construction of a loop body.

Construction of these simulations has given us considerable insight into the Gries/Dijkstra process. We have also constructed a prototype process program that supports human application of the method [39]. Ideally, we would like it to operate in a standard environment and interact with other tools; for example, in the Arcadia framework [32,33]. 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. Dijkstra, E. W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", *Communications of the ACM 18*, 8 (August 1975), 453-457.
5. Dijkstra, E. W., *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1976.
6. Fagan, M. E., "Advances in Software Inspections", *IEEE Transactions on Software Engineering SE-12, 7* (July 1986), 744-751.
7. Fairley, R., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.
8. Feather, M. S., "Constructing Specifications by Combining Parallel Elaborations", *IEEE Transactions on Software Engineering 15*, 2 (February 1989), 198-208.

9. Fickas, S. F., "Automating the Transformational Development of Software", *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1268-1277.
10. Freedman, D. P. and G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, Little, Brown and Company, Boston, 1982.
11. Freeman, P. and A. I. Wasserman, eds., *Tutorial on Software Design Techniques (fourth edition)*, IEEE Computer Society, Silver Spring, MD, 1983.
12. 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.
13. Gries, D., *The Science of Programming*, Springer-Verlag, New York, 1981.
14. *Proceedings of the First International Conference on the Software Process*, IEEE Computer Science Press, Los Alamitos, CA, October 1991.
15. *Proceedings of the 6th International Software Process Workshop*, IEEE Computer Society Press, Los Alamitos, CA, October 1990.
16. *Proceedings of the 7th International Software Process Workshop*, IEEE Computer Society Press, Los Alamitos, CA, 1991.
17. Johnson, W. L., "Deriving Specifications from Requirements", *Proceedings of the 10th International Conference on Software Engineering*, April 1988, 428-438.
18. Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
19. 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.
20. Kant, E., "Understanding and Automating Algorithm Design", *IEEE Transactions on Software Engineering SE-11, 11* (November 1985), 1361-1374.
21. Kemmerer, R. A., "Testing Formal Specifications to Detect Design Errors", *IEEE Transactions on Software Engineering SE-11, 1* (January 1985), 32-43.
22. 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.
23. Linger, R. C., H. D. Mills and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA, 1979.
24. Loeckx, J. and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, New York, 1984.
25. Mills, H. D., M. Dyer and R. Linger, "Cleanroom Software Engineering", *IEEE Software 4*, 5 (September 1987), 19-25.
26. Osterweil, L. J., "Software Processes Are Software Too", *Proceedings of the 9th International Conference on Software Engineering*, 1987, 2-13.
27. 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.
28. Rich, C. and R. C. Waters, eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufman Publishers, Los Altos, CA, 1986.
29. 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.
30. 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.
31. Smith, D. R., "KIDS: a Semiautomatic Program Development System", *IEEE Transactions on Software Engineering 16*, 9 (September 1990), 1024-1043.
32. 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.
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.
38. Terwilliger, R. B., "Simulating the Gries/Dijkstra Design Process", *Proceedings of the 7th Knowledge-Based Software Engineering Conference*, September 1992, 144-153.
39. Terwilliger, R. B., "Towards Tools to Support the Gries/Dijkstra Design Process", Report No. CU-CS-594-92, Dept. of Computer Science, U. of Colorado at Boulder, May 1992.
40. Weinberg, G. M. and D. P. Freedman, "Reviews, Walkthroughs, and Inspections", *IEEE Transactions on Software Engineering SE-10, 1* (January 1984), 68-72.
41. Wing, J. M., "A Study of 12 Specifications of the Library Problem", *IEEE Software 5*, 4 (July 1988), 66-76.
42. Yourdon, E. N., *Structured Walkthroughs*, Yourdon Press, New York, 1989.
43. *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.