Towards Tools to Support the Gries/Dijkstra Design Process

Robert B. Terwilliger

CU-CS-594-92 May 1992
Towards Tools to Support the Gries/Dijkstra Design Process

Robert B. Terwilliger

Department of Computer Science
University of Colorado
Boulder, CO 80309-0430
e-mail: '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 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. In this paper, we describe our first step towards process programming this method: ISLET, a language-oriented program/proof editor. ISLET supports simple stepwise refinement with proof by automatically generating and mechanically certifying verification conditions. In addition, through ISLET the programmer has access to a library of pre-verified cliches that can be used to create programs more easily. 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 it is 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, 25, 31, 48, 50]. 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, 30, 39, 40]. 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, 33, 35-37]. 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, 26-28]. 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 one iteration including steps one ( walkthrough), two (simulation) and three (process programming) on this method [46, 47].

As a first step towards process programming the Gries/Dijkstra method, we have constructed ISLET, a language-oriented program/proof editor that can be used to perform simple stepwise refinement with proof. In such a use, each elaboration of the partial program generates verification conditions (VCs) that must hold for the transformation to be correct. ISLET automatically generates VCs and contains tools that can mechanically certify a large percentage of them at very low cost. Through ISLET, the programmer also has access to a library of pre-verified cliches describing solutions to common programming problems. These cliches can be applied to specifications to create correct programs more easily. Although the user interface is quite primitive, ISLET is highly interactive and can be viewed as a rather simple-minded intelligent assistant.

In the remainder of this paper, we describe ISLET in more detail. In section two we give some background on the Gries/Dijkstra design process and present an example walkthrough. In section three we describe the architecture of ISLET and give an example of how it can be used to perform simple stepwise refinement with proof. In section four, we give an example of how ISLET supports a more complex, cliche-based design process, and finally, in section five 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, 49, 51]. 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 [46] 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.

```plaintext
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.

```plaintext
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. 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 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 must 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-users" to obtain the following invariant.

{inv P:users-users ∧ who_has = {u∈users-users: corec(u,b)∈checks}}

The variable "usr" holds the users still to be examined. At any point during the loops execution, "users" 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, users := {}, 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-users" is equal to "users"; therefore, the loop guard is "users≠[]", 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 "users". We formalize this as "|users|". We check that this function satisfies item four of the proof rule: "|users|" is greater than zero as long as the loop is running. We now have the following.

{Q: true}
var users : set(user) ;
who_has,users := {},users ;
{inv P:users-users ∧ who_has = {u∈users-users: corec(u,b)∈checks}}
{bnd t: |users|}
do users := {} → < unknown_1 > 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 "users". If we declare a local variable "usr" of type "user" then we can accomplish this as follows.

choose (usr,usr) ; users := users-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
\begin{itemize}
\item \texttt{Q: true} \\
\texttt{var \texttt{users} : set(user);} \\
\texttt{var \texttt{usr} : user;} \\
\texttt{who\_has, \texttt{users} := \{\}, \texttt{users};} \\
\texttt{(inv \texttt{F: users, users} \land} \\
\texttt{\quad \quad who\_has =} \\
\texttt{\quad \quad \{u \in \texttt{users} \mid \texttt{usr} \in \texttt{users},} \\
\texttt{\quad \quad \quad \texttt{corec(u,b) \in \texttt{checks}}\}} \\
\texttt{(bnd \texttt{t: [users]}} \\
\texttt{do \texttt{users} \neq \{\} \rightarrow} \\
\texttt{\quad \quad \quad \quad \texttt{choose(users, usr);} \\
\texttt{\quad \quad \quad \quad \texttt{users := users \setminus usr;}} \\
\texttt{\quad \quad \quad \quad \texttt{if corec(usr,b) \in \texttt{checks} \rightarrow} \\
\texttt{\quad \quad \quad \quad \quad \quad \texttt{who\_has := who\_has \cup \texttt{usr;}} \\
\texttt{\quad \quad \quad \quad \quad \quad \texttt{[] corec(usr,b) \notin \texttt{checks} \rightarrow}} \\
\texttt{\quad \quad \quad \quad \quad \quad \quad \quad \texttt{skip;}} \\
\texttt{\quad \quad \quad \quad fi}} \\
\texttt{od} \\
\texttt{(R: \texttt{who\_has =}} \\
\texttt{\{u \in \texttt{users:} \\
\texttt{\quad \quad \quad \quad \texttt{corec(u,b) \in \texttt{checks}}\}} \\
\end{itemize}

Figure 2. Completed Who Has Design

Figure 3 shows the architecture of ISLET. The system operates on a set of symbol tables that are stored in a module data base. The user interacts with three subsystems: the editing tool allows programs to be entered and modified, the proof tool supports the mechanical certification of verification conditions, and the cliche tool gives access to a library of cliches representing solutions to common programming problems. All of these sub-systems use an algebraic simplifier to reduce logical formulae to more compact forms.

The language-oriented editing tool is similar to [32]. It provides commands to add, delete, and refine constructs. As the program/proof is incrementally constructed, the syntax and semantics are constantly checked. When necessary, the editing tool also generates verification conditions and uses the algebraic simplifier to reduce them to a canonical form. The simplifier is implemented as a term rewriting system [24, 29] and contains a knowledge-base of rules which are assumed to be convergent; it applies rules until no further changes are produced.

3. ISLET

ISLET [42] is a language-oriented program/proof editor originally developed as part of the ENCOMPASS environment [41,43-45]. ISLET supports the construction of formal specifications and their incremental refinement into verified implementations. Some refinement steps generate verification conditions that must be true for the step to be correct. In ISLET, these VCs are first subjected to a number of simple (and inexpensive) proof tactics. If a set of VCs cannot be proved using these methods alone, then they are recorded and can later be submitted to a peer review process, a series of tests, or a more powerful (and expensive) mechanical proof system.
The proof tool performs two main classes of operations. First, it applies substitutions that might cause the simplifier to go into an infinite loop. Second, it performs more complex manipulations involving dissecting and reassembling formulae. For example, the proof tool might certify the formula "A \implies B \land C" by proving first "A \implies B", and then "A \implies C".

The cliche tool contains a library of cliches representing solutions to common programming problems. For example, a cliche might encode the fact that a loop and accumulator can be used to compute the sum of all the elements in an array. With the aid of the tool, the programmer can easily apply cliches to specifications and produce verified designs very quickly.

ISLET provides significant support for simple stepwise refinement with proof as shown in Figure 1. In this process, the programmer makes all important decisions and must generate intermediate assertions for the proof. ISLET completely automates the generation of VCs and provides ample assistance for the recording and mechanical certification of them. Using ISLET, simple stepwise refinement with proof is quite practical for small programs.

For example, consider the development of a function to compute the value of a number raised to a certain power. Using ISLET, development might proceed as follows. After entering the specification and opening the corresponding scope, the programmer would observe the following.

\begin{verbatim}
function \text{exp}(x, y: \text{natural}) : \text{natural} \text{ is}
\begin{align*}
Q & : \text{true} \\
& \langle \text{unknown} \_1 \rangle \\
\{ \text{inv P} : k \leq y \land \exp = x^k \} \\
\{ \text{bdn t} : y-k \}
\end{align*}
\begin{align*}
do & \quad k \neq y \\
& \langle Q1 : k \leq y \land \exp = x^k \land y-k = t1 \rangle \\
& \langle \text{unknown} \_2 \rangle \\
R & : \exp = x^y
\end{align*}
\end{verbatim}

The function "exp" takes two natural numbers, "x" and "y", as parameters. The pre-condition for the function is "true", and the post-condition is that the return value of the function is equal to "x" raised to the "y"th power. Note that as in Pascal, we use a variable with the same name as the function to hold the return value.

Since our design language does not contain an operator to compute exponents, we must use a loop to generate the result. We can replace the constant "y" in the post-condition with the variable "k" to obtain the following invariant.

\begin{verbatim}
\{ \text{inv P} : k \leq y \land \exp = x^k \}
\end{verbatim}

The variable "k" serves as the loop counter. At any point during the loops execution, the return value of the function is equal to "x" raised to the "k"th power. It follows directly that the loop guard is "k \neq y" and that the bound function is "y-k".

In the editing tool, a single command refines an unknown into a loop; it prompts the programmer for the loop guard, invariant and bound function. We therefore first declare the variable "k", then use the "refine" command to enter the above information. The following partial program is produced.

\begin{verbatim}
function \text{exp}(x, y: \text{natural}) : \text{natural} \text{ is}
\begin{align*}
Q & : \text{true} \\
& \langle \text{unknown} \_1 \rangle \\
\{ \text{inv P} : k \leq y \land \exp = x^k \} \\
\{ \text{bdn t} : y-k \}
\end{align*}
\begin{align*}
do & \quad k \neq y \\
& \langle Q1 : k \leq y \land \exp = x^k \land y-k = t1 \rangle \\
& \langle \text{unknown} \_2 \rangle \\
R & : \exp = x^y
\end{align*}
\end{verbatim}

This refinement must satisfy the appropriate items from the loop proof rule given in section two. The post-condition on "unknown_1" ensures that the invariant will be initialized correctly. The post-condition on "unknown_2" requires that the loop body both maintain the invariant and decrease the bound. It remains to show that the termination of the loop with the invariant true guarantees the post-condition, and that bound function is greater than zero while the loop is running. The editing tool generates verification conditions for the above and the simplifier reduces them to the following.

1) k \leq y \land z = x^k \land k = y \implies z = x^y
2) k \leq y \land z = x^k \land \text{not k = y} \implies y - k > 0

Both of these formulae can be certified using the proof tool. In the first formula, since "k=y" appears in the antecedent, the system removes the equality and substitutes "k" for "y" in the rest of the formula to produce

k \leq y \land z = x^k \implies z = x^k.

This formula is reduced to "true" by the simplifier and the first verification condition is certified.

To prove the second verification condition, the proof tool first recombines conjuncts and simplifies. This reduces "k \leq y" and "not k = y" to "k < y". Then, the system checks each conjunct of the antecedent to see if it implies any conjunct of the consequent. The simplifier is able to reduce "k < y \implies y - k > 0" to "true", and the second verification condition is certified.

We now initialize the loop by refining "unknown_1" into the simultaneous assignment "k, \exp := 0, 1". The editing tool generates the verification
condition "true => 0<=y \land 1 = x^-0", which the simplifier reduces to "true". Finally, we complete the loop body by refining "unknown_2" into the multiple assignment "k, exp:=k+1, exp*x".

The editing tool generates the verification conditions

\[ k<y \land z = x^-k \land y-k = t_1 => k+1\leq y \land z^*x = x^-k+1 \land y-(k+1) < t_1 \]

which the simplifier reduces to

\[ k<y \land z = x^-k => k+1\leq y \land z^*x = x^-k+1 \]

The proof tool is then invoked, and the system proves each conjunct of the consequent separately. Substituting "x^k" for "z" generates the formula "k<y => x^k+1 \land z^*x = x^-k+1" which the simplifier reduces to "true". The proof tool rearranges conjuncts to produce the formula "k<y => k+1\leq y" which can be reduced to "true" by the simplifier. Since both conjuncts of the consequent have been proven, the original verification condition is certified.

In the above example, stepwise refinement with proof was used to transform a specification into a verified design. The programmer made all important decisions and had to generate intermediate assertions for the proof. While this process has a certain elegance, it is difficult to extend to larger and more complex designs. As the programs become more complex, generating the intermediate assertions becomes more and more of a burden. Unfortunately, the point of infeasibility is reached quite quickly; however, using pre-verified cliches more complex designs can be generated quite easily.

4. Cliche-Based Development

Figure 4 shows a pictorial representation of the cliche-based design process supported by ISLET. It has two levels. At the lower level, a programmer transforms formal specifications into correct designs using a library of cliches representing solutions to common programming problems. On the upper level, a master designer 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 directly supported by the cliche tool, while the rest is performed using traditional methods.

The cliche tool gives a programmer access to the library of pre-verified cliches. Each cliche has an applicability condition, as well as a rule for transforming specifications into more complete programs. When the programmer instructs the tool to apply cliches to a particular specification, 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 "undo" command 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, cliche-based architecture is that proofs are performed mostly at "compile" rather than "run" time. Cliche construction and verification are quite difficult, but are done only once for each cliche and performed by a master designer. On the other hand cliche application is quite easy using ISLET, and is performed repeatedly by the programmer during development.

This division of labor allows verified programs to be produced very quickly by persons reasonably skilled in the use of formal methods. The programmer's task is reduced to handling the cases for which the library
contains no applicable cliches. As the library grows with experience, this situation should occur less and less often. Also, as a programmer's familiarity with the library grows, their ability to handle such cases should increase. We believe that this cliche-based development process is in tune with human's natural work strategies; as we understand it, this is supported by work on the psychological aspects of programming [4, 5, 34, 38].

For example, let us reconsider the "who_has" function presented in section two.

\[(Q: \text{a.staff})\]
\[< \text{unknown}_0 > ;\]
\[\{R: \text{who}_\text{has} = \{u\in\text{users}: \text{corec}(u,b)\in\text{checks}\}\}\]

Figure 5 shows a simplified representation of the "conditionalIteration_on_set" cliche which is applicable to this specification. Instantiations of this cliche solve problems using a loop with an embedded conditional. The post-condition of the cliche states that the result variable, "Var", is equal to the value of "Iop(\text{Set},\text{Cond})"; in other words, to the value of an iteration operator applied to a set with a certain condition.

The body of "conditionalIteration_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 iterates over all the items in "Set". If the item in question satisfies "Cond" then "Var" is set to "Op(\text{Var},\text{Lvar})"; otherwise, nothing is done. 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.

\[
\{Q: \text{true}\}
\]
\[\{\text{var} \text{users : set(user)}\};\]
\[\{\text{var} \text{usr : user}\};\]
\[\{\text{var} \text{who}_\text{has}, \text{users} := \{} \{\}, \text{users}\};\]
\[\{\text{inv} P: \text{users} \subseteq\text{users} \wedge \text{who}_\text{has} = \{u\in\text{users}: \text{corec}(u,b)\in\text{checks}\}\}\]

Figure 5. Conditional Iteration on Set Cliche

Figure 6. Instantiated Loop Cliche
1) \( \text{Id} = \text{iop}([], \text{Cond}) \)

2.1) \((s \in S \land \text{Cond}(s) \Rightarrow \text{iop}(s, \text{Cond}) = \text{Op}(\text{iop}(s-s, \text{Cond}), s)) \)

2.2) \((s \in S \land \neg \text{Cond}(s) \Rightarrow \text{iop}(s, \text{Cond}) = \text{iop}(s-s, \text{Cond})) \)

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.

In our example, "iop_table" contains the entry "( {}, \text{Var}+\text{Var}, [s \in \text{Set}: \text{Cond}(s)] ) ", and Figure 6 shows the result of applying the cliche to the "\text{who}\_\text{has}" specification. The overall structure of the design is now evident. The loop iterates over all the users in the library. The variable "\text{usr}" holds the set of all users still to be considered, while "\text{usr}\_\text{current}\_\text{being}\_\text{examined}" holds the user currently being examined.

The loop body must now be completed. Below is a simplified representation of the "simple\_if\_then\_else" cliche which is applicable to this problem.

```
cliche simple_if_then_else is
  {Q}
  if B1 \rightarrow \{ Q \land B1 \} \land S1 > \{ B1 \land E1 \}
     \[ B2 \rightarrow \{ Q \land B2 \} \land S2 > \{ B2 \land E2 \} \]
  fi
  \{ R: B1 \land E1 \lor B2 \land E2 \}
if
  is_negation(B1, B2) ;
end simple_if_then_else ;
```

This cliche says that the statement "if \( B1 \rightarrow S1 \) \& \( S2 \) fi" is correct with respect to a pre-condition "Q" and post-condition "\( B1 \land E1 \lor B2 \land E2 \)" if: "\( B1 \)" is the logical negation of "\( B2 \)"; "\( S1 \)" is correct with respect to pre- and post-conditions "\( Q \land B1 \)" and "\( B1 \land E1 \)" respectively; and "\( S2 \)" is correct with respect to "\( Q \land B2 \)" and "\( B2 \land E2 \)".

Application of "simple\_if\_then\_else" generates the following design for the loop body.

```
if \text{corec}(\text{usr}, b) @ \text{checks} \rightarrow
  \{ Q2: \text{who}\_\text{has}=\text{WHO}\_\text{HAS} \land
    \text{corec}(\text{usr}, b) @ \text{checks} \}
  < \text{unknown}_2 > ;
  \{ R2: \text{corec}(\text{usr}, b) @ \text{checks} \land
    \text{who}\_\text{has}=\text{WHO}\_\text{HAS} \land
    \text{corec}(\text{usr}, b) @ \text{checks} \}
  \}
\fi
```

If the user in question has the book checked out, then "\text{who}\_\text{has}" must be set to its initial value plus that user. If the user does not have the book checked out, then the value of "\text{who}\_\text{has}" is not changed.

Below is a simplified representation of the "simple\_assignment" cliche, which is applicable to "unknown\_2" above.

```
cliche simple\_assignment is
  \{ Q \} \text{Var}_1..\text{Var}_n := \text{Solv}_1..\text{Solv}_n \{ R \}
  if
    Q \Rightarrow R[\text{Var}_1..\text{Var}_n \setminus \text{Solv}_1..\text{Solv}_n]
  end simple\_assignment ;
```

This cliche states that the statement "\text{Var}_1..\text{Var}_n := \text{Solv}_1..\text{Solv}_n" is correct with respect to a pre-condition "Q" and post-condition "R" if "Q" implies "R" with "\text{Solv}_1..\text{Solv}_n" substituted for "\text{Var}_1..\text{Var}_n".

The "do\_nothing" cliche is applicable to "unknown\_3".

```
cliche do\_nothing is
  \{ Q \} \text{skip} \{ R \}
  if
    Q \Rightarrow R
  end do\_nothing ;
```

This cliche states that the null statement, "\text{skip}", satisfies a specification if the pre-condition implies that the post-condition is already true.

Applying the "simple\_assignment" and "do\_nothing" cliches to "unknown\_2" and "unknown\_3" respectively produces the following.

```
if \text{corec}(\text{usr}, b) @ \text{checks} \rightarrow
  \text{who}\_\text{has}=\text{who}\_\text{has}+\text{usr} ;
\[ \text{corec}(\text{usr}, b) @ \text{checks} \rightarrow \text{skip} ; \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 "\text{who}\_\text{has}" if not then nothing is done. The cliche-based design process has now produced the complete design shown in Figure 2.

The number of cliches that can be used in the design process is literally infinite, but we were able produce a reasonable design using only four (these cliches are presented in more detail in [47]). As the number of cliches increases, certain problems with the current process become more significant. For example, at present the library is searched in a fixed order and the first applicable cliche is used. More realistically, if
more than one cliche is applicable the programmer might want to examine them all and then decide which is most appropriate. While the cliche tool can easily be modified to support this interaction, it may require considerable expertise to understand and evaluate cliches written by someone else.

Despite this and other drawbacks, we are pleased with the cliched-based design process as supported by ISLET. While definitely not the final answer, it does provide an initial step towards tools to support the Gries/Dijkstra design process.

5. 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 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 one iteration including steps one (walkthrough), two (simulation), and three (process programming) on this method [46,47]. 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.

As a first step towards process programming, we have constructed ISLET, a prototype tool to support the Gries/Dijkstra method. The system is based on a language-oriented program/proof editor and uses a library of cliches describing solutions to common programming problems. In the simplest case, ISLET can be used to perform simple stepwise refinement with proof; each elaboration of the partial program generates verification conditions (VCs) that must hold for the transformation to be correct. ISLET automatically generates VCs for refinements and contains tools that can mechanically certify a large percentage of these VCs at very low cost.

In this simple design process, the programmer makes all the design decisions and must generate intermediate assertions for the proofs. While this method has a certain elegance, it is difficult to extend to larger and more complex designs. As the programs become more complex, generating the intermediate assertions necessary for the proofs becomes more and more of a burden. Unfortunately, the point of infeasibility is reached quite quickly; therefore, ISLET also supports the application of pre-verified cliches to create programs more easily.

The cliche-based design process supported by ISLET has two levels. At the lower level, a programmer transforms formal specifications into correct designs using a library of cliches. On the upper level, a master designer uses strategies and proof rules to construct and verify cliches for the library. The advantage of our two level architecture is that proofs are performed mostly at "compile" rather than "run" time. Cliche construction and verification are quite difficult, but are done only once for each cliche and performed by a master designer. On the other hand using ISLET, cliche application is reasonably easy and is performed repeatedly by the programmer during development.

A prototype implementation has been written in Prolog and used to generate complete designs for several small examples including Kemmerer's Library Problem. We are currently considering a more robust implementation. Ideally, we would like ISLET to operate in a standard environment and interact with other tools; for example, in the Arcadia framework [39,40]. 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.

6. References

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.