Integrating Three Tool-Based Approaches to Software Engineering

Robin A. Nicholl

Department of Computer Science,
University of Western Ontario,
London, CANADA N6A 5B7

ABSTRACT
The need for software which is more reliable and more easily maintained, together with the increasing costs of software development and maintenance, have led to greater use of software tools to support software engineering activities. Moreover, some of the approaches once considered uneconomic, such as program transformation and program verification, now appear worthy of greater attention. We describe a software tool which supports transformation and verification activities, as well as some more widely used dynamic analysis activities. We explain how the design of this tool supports not only the different activities but also their interaction, and the advantages of integrating these activities are briefly discussed. We report on the current status of our implementation and on enhancements which are currently being implemented.

1. Introduction
The need to develop and maintain reliable software systems for ever more complex problems has increased interest in the development and use of sophisticated software tools. Economic considerations also suggest that software tools should be used to assist in the processes of software development and maintenance. Experience with the use of individual software tools has indicated that even greater benefits will result from adapting them to work together. This has driven current research into integrated program support environments and software engineering environments [11], some of which are designed to support specific software development methodologies, while others are independent of methodology.

Such software tools may be divided into two classes. Some tools keep track of documents created during the software life-cycle, but are ignorant of the structure or content of those documents. Other tools examine the structure or content of such documents to carry out some form of analysis, and in some cases they may alter the documents to meet some criterion. We describe the integration of a number of tools of this latter class, although in the future we would hope to combine these with tools of the former class.

The specific approaches which we have combined into a single software tool are program transformation, program verification and dynamic analysis. Their integration is achieved through a common user interface and a shared program representation — a simple parse tree. To promote interaction between the tools, care has been taken to decompose more complex tools for each distinct approach into several smaller tools which either present information to the user or update the parse tree, thus making the output of one tool immediately available as the input to another.

We will now discuss related work in the areas of program transformation, program verification and dynamic analysis, and in the development of program models. We will then describe the language used to express programs to be manipulated, before explaining in detail what types of manipulation are provided. Our implementation will then be described, and finally we will present our conclusions concerning the usefulness of this approach and indicate directions in which future work should proceed.

2. Background
We briefly discuss related work in each of the areas which are brought together in our approach. A considerable amount of work has been done on program transformation and on program verification. Our discussion of these areas concentrates on software tool support and on well-known or commonly used techniques.

2.1. Program transformation
Program transformation is based on the observation that, although there are many different programs to solve the same problem, properties of programming languages will allow us to demonstrate that some of these programs are equivalent. Since this equivalence depends on properties
of the programming language, we should expect to be able to take programs satisfying one property and transform them into equivalent programs satisfying another property.

Some authors appear to advocate program transformation as an automated approach for constructing correct and efficient programs from their "specifications" as very high level and inefficient programs. A more pragmatic approach (illustrated by [3]) is to require problem-specific transformation rules to be written by a software developer, to be added to the set of general purpose transformation rules implemented by the transformation system.

The most extensive use of transformational programming appears to be the CIP-L project at the Technical University of Munich [1]. This project is based on a "wide-spectrum" language which is to be used to express a variety of programming styles. Some of these styles correspond to (formal) specifications while others correspond to efficient (conventional) programs. Program transformations may be used to transform programs from one style to another, without affecting their logical behaviour. Partsch and Steinbruggen [10] present a thorough survey of work on program transformation systems.

2.2. Program Verification

Program verification is based on the observation that the effect of programs written in a (well-defined) programming language is given by the language definition. Hence we may overcome the limitations of program testing if we can prove that the defined effect of a program satisfies the program specification. Inevitably work on program verification has been closely related to work on the definition of programming languages, such as the axiomatic definition of Pascal [6]. Different approaches to verification are surveyed in [8].

The typical software tool to support program verification is the verification condition generator. A verification condition generator takes as input a program to which have been added logical assertions describing conditions which must hold at particular points in the program's execution. These conditions incorporate the behaviour required by the program specification. The output of the verification condition generator is a list of logical theorems (verification conditions) to be proven. If all of the theorems are true then the logical assertions are all true and hence the program meets its specification. Gray [4] gives a detailed description of the construction of such a tool. An automated theorem prover is sometimes applied to the generated verification conditions, to reduce the manual effort involved in proving that they are true.

Although it is no longer considered practical to prove an existing program correct, it is quite practical to use proof of correctness ideas to guide the programming process [5, p. 164]. Verification condition generators may assist in this process, and in proving some properties of a program's behaviour.

2.3. Dynamic Analysis

Dynamic analysis is based on the observation that much information about the behaviour of a program is most readily available while the program is executing. Since programs do not generally report their internal behaviour, some changes must be made to them before this information can be obtained. One widely used dynamic analysis tool is the performance profiler (such as prof under Unix [2]), which provides information such as the time spent executing different parts of a program or the number of times each statement has been executed.

Dynamic analysis tools have also been used to monitor the logical behaviour of programs. In this case logical assertions are inserted into the program text, and these are evaluated during execution of the program. If any of these assertions is found to be false, some indication is given to the programmer that something is wrong. A comprehensive description of one such tool is given by Stucki [12]. Dynamic analysis is a fully automated process, in which a program is input to a software tool which then constructs a program with the extended functional behaviour that it reports on its own execution behaviour. No attempt is ever made to prove that assertions are true - they are simply evaluated for the particular values assumed by program variables during execution.

2.4. Program Models

Program models are used to present some program property in a manner which should be easy to understand or efficient to manipulate: flowcharts and parse trees being probably the best known examples. Since different software tools make use of different properties of programs, they usually operate most efficiently on different representations (or models) of programs. The current trend towards integrated software tools implies a need for more general program models, even though individual tools may execute more slowly as a result. The most extensively used general model is the attributed parse tree: a parse tree of the program whose nodes record additional information on non-syntactic properties of programs.

3. Integrating the Three Approaches

We will now describe a sample input language to support the three approaches in an integrated fashion. We then describe activities in each of the three areas and discuss
how they interact with each other via a shared program
model.

3.1. The Input Language
To provide for verification and for dynamic analysis it is
necessary to extend a basic programming language to
include logical assertions. Examples of such language
extensions are widely available throughout the literature
on program verification and dynamic analysis [4,11]. We
will concentrate on the language accepted by our current
implementation and trust that the reader will see how to
similarly extend other programming languages.

The programming language used is a subset of Modula-2
[13]. The major omissions are data structures (arrays are
partially supported, other data structures are not supported
at all) and some control constructs. Extensions to the
language are logical assertions, in a variety of forms, and
multiple assignment statements. Modula-2 language
features of the input language are: declarations of
modules, procedures, constants, types and variables;
import and export lists; and assignments, procedure calls,
if statements and while statements. The additional
features of the input language are now described in detail.

1. Multiple assignment statements. These have the form
variable "=" variable
expression "=" expression

and have the effect of performing several simple
assignment statements at the same time. This is most
simply illustrated by the following multiple assign-
ment to exchange the values of x and y

\[ x, y := y, x \]

These multiple assignment statements were introduced to
support our program transformations, but they have also
proved useful for program verification.

2. Logical assertions. Four types of logical assertions
may be used: ENTRY assertions, EXIT assertions,
ASSERT assertions and INVARIANT assertions. Each
has the form of a keyword, to identify the type of
assertion, followed by a logical expression. ENTRY
and EXIT assertions must be true at the beginning
and end (respectively) of the procedure or module in
which they appear. ASSERT assertions must be true at
the point in the statement list where they appear.
INVARIANT assertions must be true at several points
in the execution of the loop with which they are asso-
ciated.

A procedure to compute factorial may be expressed in
this language as shown in Figure 1. It includes references
to a function \texttt{factorial} which will act as the definition
of factorial, and is considered to be part of the specification

\begin{verbatim}
PROCEDURE ComputeFactorial
(n: INTEGER; VAR fac: INTEGER);
ENTRY n >= 0;
EXIT fac = factorial(n);
VAR i, f: INTEGER;
BEGIN i := 0; f := 1;
INVARIANT f = factorial(i) & i <= n;
WHILE i <= n DO
  i := i + 1; f := f * i
END;
fac := f
END ComputeFactorial;
\end{verbatim}

Figure 1. Factorial procedure: input text.

3.2. Transformation Activities
The program transformations used are selected from the
equivalences called "Laws of Programming" [7]. The
transformations available will change the positions of
assignment statements, replace procedure calls by
(appropriately transformed) procedure bodies, and intro-
duce or eliminate multiple assignment statements. Per-
forming these transformations will always result in the
creation of an equivalent program.

The following laws from [7] are included. Let x and y be
lists of distinct variable names, and let E and F denote
lists of expressions. Let G(x) denote a list of expressions
involving the variables of x.

\begin{enumerate}
  \item \((x := E; y := F) = (x,y := E,F)\)
    \quad if no variable in x appears in F.
  \item \((x := E; y := G(x)) = (x,y := E,G(E))\)
  \item \((x,y := E,y) = (x := E)\)
  \item \((x := E; IF p(x) THEN S1 ELSE S2 END) =
               (IF p(E) THEN x := E; S1
               ELSE x := E; S2 END)\)
  \item \((IF p THEN S1 ELSE S2 END; x := E) =
               (IF p THEN S1; x := E
               ELSE S2; x := E END)\)
\end{enumerate}

These laws are illustrated by the following examples.

\begin{enumerate}
  \item \((i := 0; f := 1) = (i,f := 0,1)\)
  \item \((i := i+1; f := f*i) = (i,f := i+1,f*i+1)\)
\end{enumerate}
When applied to the factorial procedure of Figure 1 these laws may give rise to the transformed version of the procedure shown in Figure 2, in which sequences of assignments have been replaced by multiple assignments.

Figure 2. Factorial procedure: transformed version.

Figure 3. Factorial procedure: transformation of a procedure call.
(1) \(a := -x\); \textbf{ASSERT} \(a = \text{ABS}(x)\) 
\textbf{⇒} \textbf{ASSERT} \(-x = \text{ABS}(x); a := -x\)

(2) \(\text{IF } x > 0 \text{ THEN } \text{ASSERT } x = \text{ABS}(x); a := x\) 
\text{ELSE} \textbf{ASSERT} \(-x = \text{ABS}(x); a := -x \text{ END}\) 
\textbf{⇒} \textbf{ASSERT} \((x > 0 \& \& x = \text{ABS}(x))\) 
\textbf{OR} \((x <= 0 \& \& -x = \text{ABS}(x)))\) 
\textbf{IF} \(x > 0 \text{ THEN } a := x \text{ ELSE } a := -x \text{ END}\)

(3) \textbf{INVARIANT} \(s = \text{sum}(i) \& i <= n; \) 
\text{WHILE} \(i \neq n \text{ DO } i, s := i + 1, s + i + 1 \text{ END}\) 
\textbf{⇒} \textbf{ASSERT} \(s = \text{sum}(i) \& i <= n; \) 
\textbf{WHILE} \(i \neq n \text{ DO}\) 
\textbf{ASSERT} \(s = \text{sum}(i) \& i <= n \text{ END}\) 
\textbf{ASSERT} \(s = \text{sum}(i) \& i <= n \& i = n; \)

(4) \textbf{ENTRY TRUE; EXIT } a = \text{ABS}(x); \) 
\textbf{IF} \(x > 0 \text{ THEN } a := x \text{ ELSE } a := -x \text{ END}\) 
\textbf{⇒} \textbf{ASSERT TRUE; } 
\textbf{IF} \(x > 0 \text{ THEN } a := x \text{ ELSE } a := -x \text{ END;} \) 
\textbf{ASSERT } a = \text{ABS}(x)\)

For other statements (such as procedure calls) for which such rules have not yet been implemented, the assertion is simply not moved and that part of the program is not changed. Hence in all cases the process of moving assertions does not affect the validity of the assertions; if they are moved they are modified to take account of their new position in the program.

3.3.2. Reporting Verification Conditions

When verification conditions are being reported, no attempt is made to move assertions around in the program, although it should normally be expected that assertions will have been moved before verification conditions are requested. To produce the report we must identify where some assertion has been moved adjacent to an \textbf{ENTRY} assertion or \textbf{INVARIANT} assertion. Whenever this occurs we report as a verification condition that the \textbf{ENTRY} or \textbf{INVARIANT} assertion (\& looping condition) must imply the adjacent \textbf{ASSERT} assertion. Wherever an \textbf{ENTRY} or \textbf{INVARIANT} assertion has no such adjacent assertion we report the omission – as a section of the program where no attempt has been made to verify correctness.

For the factorial procedure of Figure 1 the effect of moving assertions is shown in Figure 4. The \textbf{EXIT} assertion is placed after the last statement of the procedure as an \textbf{ASSERT} assertion, and is then moved back to become \textbf{ASSERT} \(f = \text{factorial}(n)\). Similarly the \textbf{INVARIANT} assertion is placed at the relevant points of the procedure as an \textbf{ASSERT} assertion and is then moved back. If we

\textbf{PROCEDURE} ComputeFactorial 
\(n: \text{INTEGER}; \text{VAR } f: \text{INTEGER}; \) 
\textbf{ENTRY} n := 0; 
\textbf{EXIT} f := factorial(n); 
\textbf{VAR} i, f: INTEGER; 
\textbf{BEGIN } \text{assert } 1 = \text{factorial}(0) \& \& 0 <= n; 
i := 0; 
\textbf{INVARIANT} f = \text{factorial}(i) \& \& i <= n; 
\textbf{WHILE} i <= n \text{ DO } 
\textbf{ASSERT} f \& (i + 1) = \text{factorial}(i + 1) \& \& (i + 1) <= n; 
i := i + 1; f := f * i; 
\textbf{END}; 
\textbf{ASSERT} f = \text{factorial}(n); 
fac := f 
\textbf{END} \text{ComputeFactorial}; \)

\textbf{Figure 4. Factorial procedure: assertions moved.}

ask for the verification conditions of Figure 4 to be reported we obtain:

(1) \textbf{ENTRY assertion} 
\(n <= 0 \Rightarrow 1 = \text{factorial}(0) \& \& 0 <= n.\)

(2) \textbf{Loop body entry} 
\(f = \text{factorial}(i) \& \& i <= n \& \& i <= n \) 
\textbf{⇒} \(f \& (i + 1) = \text{factorial}(i + 1) \& \& (i + 1) <= n.\)

(3) \textbf{Loop exit} 
\(f = \text{factorial}(i) \& \& i <= n \& \& i = n \) 
\textbf{⇒} \(f = \text{factorial}(n).\)

However if instead we ask for a report of the verification conditions for Figure 1 we will be informed that a complete list of verification conditions is not available; in fact there are none to report – since in Figure 1 the preparatory task of moving assertions has not yet been done.

3.4. Dynamic analysis activities

Current facilities for dynamic analysis are limited to checking the truth of logical assertions during program execution. For each type of assertion (\textbf{ENTRY, EXIT, INVARIANT} and \textbf{ASSERT}) additional code is inserted at the appropriate point in the program. This code will evaluate the assertion and report a message each time the assertion is false.

If we request dynamic analysis of the factorial procedure of Figure 1, the procedure will be changed into the form shown in Figure 5. Execution of this version of the procedure will provide the necessary information on the (internal) behaviour of the procedure. Of course in order to execute this version our definition of the \textit{function factorial}, which is used in several assertions, must be execut-
able.

PROCEDURE ComputeFactorial
   (n: INTEGER; VAR fac: INTEGER);
VAR i, f: INTEGER;
BEGIN
   IF NOT (n >= 0) THEN ... END;
   i := 0; f := 1;
   IF NOT (i = factorial(n) & i <= n) THEN ... END;
   WHILE i <= n DO
      i := i + 1; f := f * i;
      IF NOT (f = factorial(i) & i <= n) THEN ... END;
   END;
   fac := f;
   IF NOT (fac = factorial(n)) THEN ... END;
END ComputeFactorial;

Figure 5. Factorial procedure:
   for dynamic analysis (outline).

4. Implementation and Analysis of the Approach to Integration

Our implementation of this integrated tool is written in Modula-2 and runs on SUN-3/50s under Unix. It consists of a single main module plus 13 separate modules – 27 files in all, containing a total of 7,281 lines of text. Figure 6 summarises the types and sizes of files making up the system. The largest modules are Transform (1,639 lines), Parser (1,167 lines) and Unparser (1,141 lines). The sizes of these three modules (and of a number of smaller modules) varies with the size of the input language.

<table>
<thead>
<tr>
<th>File Type</th>
<th>min</th>
<th>max</th>
<th>mean</th>
<th>median</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interfaces (13)</td>
<td>20</td>
<td>37</td>
<td>41</td>
<td>42</td>
<td>524</td>
</tr>
<tr>
<td>Implementations (14)</td>
<td>19</td>
<td>1,639</td>
<td>482</td>
<td>228</td>
<td>6,747</td>
</tr>
</tbody>
</table>

Figure 6. Number of lines per source file.

The structure of this system is shown in Figure 7, and we would expect most designers to choose a broadly similar structure. The activities of transformation, verification and dynamic analysis are performed independently and can be selected individually by the user. Translation between program texts and program models is performed by the parser and the "unparser". In our current implementation the unparser can display the model in a variety of different formats: exactly as stored (with assertions); in a form for compilation (with assertions as comments); and in a form for dynamic analysis (with assertions as code for assertion checking). For the last two of these options the user will probably want first to expand multiple assignments into several simple assignments, to guarantee that the output will satisfy Modula-2 syntax rules.

The user interface is a simple menu selection:

1) Parse a program.
2) Unparse a program.
3) Move assertions (to compute pre-conditions).
4) Report verification conditions.
5) Transform.
6) Remove multiple assignments.

The system also allows the user to save and restore program models in an internal form. This can be used to avoid the overhead of reparsing a program text.

The program model is a parse tree. Each non-leaf node of the tree represents a syntactic construct of the language. It has a tag identifying the syntax construct and a list of children. Each leaf node is a token of the input language. The parse tree is based on the abstract syntax rather than the concrete syntax of the language, in the sense that most of the special tokens of the language, such as "WHILE", "=" and ";", are not stored in the tree. Their presence can be inferred from the tag of each node, tags such as 
   
   While-statement, assignment and statement-list. Since the tree stores no context-sensitive information, some tree traversal is necessary when we wish to obtain information such as the type of a variable. The inefficiency inherent in this approach has not proved to be a problem, and we enjoy the benefits of a simple program representation. It may become necessary to complicate this representation as the system develops further.

This software was initially developed as a vehicle for investigating and teaching some existing approaches to software engineering. The decision to incorporate different approaches in a single program was made to reduce the implementation effort, but the approaches were kept separate within the program to allow for future enhancements. The program model was kept as simple as possible (and was written as a separate module) to allow for ease of modification and because there seemed no solid basis for choosing among the different ways to complicate it.

Our ability to promote interaction among verification, transformation and dynamic analysis stems from these decisions to "keep it simple" and from the adoption of the following guidelines.

1. There are two types of tool involved: those which update the program model and those which report information to the user concerning the current state of the model. Tools which attempt to do both (such as the typical verification condition generator) should be decomposed into a number of simpler tools of each
2. It should be possible to apply tools selectively; users must be able to select particular parts of the program to which to apply individual tools.

3. All tools must operate on all states of the program model; they may not assume that some other tool has previously transformed the program model into any specific form. Such tools will make as much progress as they can on the selected portions of the program, and may not fail upon encountering an unexpected syntax construct in the program.

As a result, all tools are applicable at all times (although some times may be more appropriate than others). The tools are thus also quite robust and less vulnerable to enhancements made to other tools. The user may apply any tools in any order to any parts of the program and expect to see appropriate results. Forms of analysis normally associated with one type of update activity are now made available to all. However, the responsibility for controlling this wide range of activities rests with the user. The system supports integration of the various activities but does not provide an integrated approach itself.

We consider the resulting system to reflect a fairly pragmatic view of the three software engineering approaches incorporated in it. Users may choose to use only one approach and ignore the others, to use one approach to complement another, or to use one approach to check on another. Users may choose to work on an entire program or only on parts of a program. Assertions may be used as comments, as conditions to be tested or as conditions to be proved. The use of assertions is always optional. All tools will take them into account when they are present, but none insists upon their presence.

One of the insights gained in the design of this system is that the rules used in program verification for moving assertions may be thought of as program transformations which move assertions backwards through the program (or the program forwards through assertions). The transformations described in this paper were used manually in the development of a new two-dimensional line clipping algorithm [9], before the current implementation was complete. The algorithm takes advantage of geometric symmetry properties to allow different cases to be treated similarly – using geometric transformations to transform one case into another. Since these geometric transformations are all implemented as sequences of assignment statements, the rules given above for moving assignments were used to transform the algorithm into a more efficient program.

5. Conclusions and Future Work

We have shown how three different approaches to software engineering can be used together as parts of a single software tool. We have discussed the advantages of such integration and explained how to achieve it. An implementation of this integrated tool has been completed.

There are two main areas where future work is required: extending the input language to include a larger subset of Modula-2 and a more powerful language for expressing assertions; and extending the range of functions available to the user. The problems of extending the language subset have been faced (and mostly solved) by designers and implementers of existing tools, so the work we need to do in this area is primarily implementation of known techniques. The problems of extending the range of functions are less obvious. We would like to extend the dynamic analysis capabilities of the system: work is currently in progress to provide some performance statistics. We would also like to bridge the gap between the more formal approaches of transformation and verification and the more pragmatic approaches of dynamic analysis and testing. To this end we recently incorporated a symbolic execution facility into the system. There are also some advantages in incorporating an editor into the system. This editor should operate directly on the program model, thus avoiding the present need to switch between use of the system and use of the system editor – with the need also to parse and unparse the program.

Finally we are considering how this system can be combined with facilities for configuration management and revision control. While this appears to be essential if the system is to work on large programs, we have yet to determine the best way to integrate the different ideas.

ACKNOWLEDGEMENTS

The system described in this paper was implemented chiefly by Ed Sendek, and he has also extended the current system. Yuk-Ngo Cheng, Raymond Mak and Selina Wong have also contributed to its development. Suggestions made by the referees have led to improvements in the presentation of these results. Financial support for this research was provided by the National Science and Engineering Research Council (NSERC) of Canada under Grant No. S265A1.

REFERENCES


Figure 7. Structure of the integrated tool.