1995

The verification of object-oriented programs and programs with exception handling constructs.

Matthew. Blain
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

This online database contains the full-text of PhD dissertations and Masters’ theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
THE VERIFICATION OF OBJECT-ORIENTED PROGRAMS AND PROGRAMS WITH EXCEPTION HANDLING CONSTRUCTS

by
Matthew Blain

A Thesis
Submitted to the Faculty of Graduate Studies and Research through the School of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science at the University of Windsor

Windsor, Ontario, Canada
1995
THE AUTHOR HAS GRANTED AN IRREVOCABLE NON-EXCLUSIVE LICENCE ALLOWING THE NATIONAL LIBRARY OF CANADA TO REPRODUCE, LOAN, DISTRIBUTE OR SELL COPIES OF HIS/HER THESIS BY ANY MEANS AND IN ANY FORM OR FORMAT, MAKING THIS THESIS AVAILABLE TO INTERESTED PERSONS.

THE AUTHOR RETAINS OWNERSHIP OF THE COPYRIGHT IN HIS/HER THESIS. NEITHER THE THESIS NOR SUBSTANTIAL EXTRACTS FROM IT MAY BE PRINTED OR OTHERWISE REPRODUCED WITHOUT HIS/HER PERMISSION.

L'AUTEUR A ACCORDE UNE LICENCE IRREVOCABLE ET NON EXCLUSIVE PERMETTANT A LA BIBLIOTHEQUE NATIONALE DU CANADA DE REPRODUIRE, PRETER, DISTRIBUER OU VENDRE DES COPIES DE SA THESE DE QUELQUE MANIERE ET SOUS QUELQUE FORME QUE CE SOIT POUR METTRE DES EXEMPLAIRES DE CETTE THESE A LA DISPOSITION DES PERSONNE INTERESSEES.

L'AUTEUR CONSERVE LA PROPRIETE DU DROIT D'AUTEUR QUI PROTEGE SA THESE. NI LA THESE NI DES EXTRAITS SUBSTANTIELS DE CELLE-CI NE DOIVENT ETRE IMPRIMES OU AUTREMENT REPRODUITS SANS SON AUTORISATION.

ISBN 0-612-01432-0
Abstract

This is a masters thesis on the verification of object-oriented programs. An object-oriented mini-language and proof system are constructed. Polymorphism, one of whose characteristics is the ability to assign objects of more than one type to a given variable, is a major feature in this mini-language. Class specifications are introduced as a tool for verifying properties of classes within a program. Specifications for classes are especially valuable in the verification of reusable classes. Another problem that is addressed is how a specification for a class $B$ can be derived from a valid specification for another class $A$ given that $B$ inherits from $A$. A second mini-language and proof system are introduced for the verification of programs with exception handling constructs. Assertion violations and expression exceptions are some of the exceptions considered.
To my Parents
Acknowledgments

The author wishes to acknowledge the assistance of Dr. Liwu Li for his many suggestions on the thesis work and on the thesis subject matter.
# Contents

Abstract ................................................................. iv  
Acknowledgments ......................................................... vi  

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th>INTRODUCTION</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Object-Oriented Programming and Software</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Quality</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Verification vs Testing</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Overview of thesis</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2</th>
<th>OBJECT-ORIENTED PROGRAMMING</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LANGUAGES</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Objects</td>
</tr>
<tr>
<td>3</td>
<td>Classes</td>
</tr>
<tr>
<td>4</td>
<td>Inheritance</td>
</tr>
<tr>
<td>5</td>
<td>Polymorphism and Dynamic Binding</td>
</tr>
<tr>
<td>6</td>
<td>Generic classes</td>
</tr>
<tr>
<td>7</td>
<td>A final note</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 3</th>
<th>PROGRAM VERIFICATION</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>15</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>VERIFICATION OF PROGRAMS WITH POLYMORPHISM</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>The state of an object-oriented program</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Preliminary</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>The Mini-language POLYPLW</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Proof System for POLYPLW</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Examples</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Termination</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Soundness of Proof System for POLYPLW</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Aliased Objects</td>
<td></td>
</tr>
</tbody>
</table>

| 2 | Defining programs .................................................. 15 |
| 3 | Specifications ..................................................... 17 |
| 4 | Partial and Total Correctness ................................. 18 |
| 5 | Termination ........................................................ 18 |
| 6 | Soundness and Completeness ................................. 19 |
| 7 | Induction .......................................................... 20 |
| 8 | The language PLW ............................................... 20 |
| 9 | Example .......................................................... 22 |
Chapter 5  VERIFICATION USING CLASS

SPECIFICATIONS ................................. 56

1 Introduction ................................. 56

2 Class specifications .......................... 56

3 Sending Messages to Self .................... 59

4 Class specifications and Inheritance ........ 60

5 Class Invariant ............................... 62

6 Property Dependence ....................... 64

7 Example ..................................... 66

Chapter 6  VERIFICATION OF PROGRAMS WITH

EXCEPTION HANDLING ......................... 76

1 Introduction ................................. 76

2 Preliminaries ............................... 77

3 The mini-language EPLW .................... 81

4 Proof System for EPLW ..................... 84

5 Soundness of Proof System .................. 89

6 Exceptions in Expressions ................... 96

7 Termination ................................. 99

8 Object-Oriented Extentions ................. 101

9 Non-determinism and Exceptions .......... 104

10 Exception handling in real languages .... 106
<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>CONVERSION</th>
<th>113</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>113</td>
</tr>
<tr>
<td>2</td>
<td>Generic Classes</td>
<td>113</td>
</tr>
<tr>
<td>3</td>
<td>&quot;Once&quot; routines</td>
<td>117</td>
</tr>
<tr>
<td>4</td>
<td>Constructs within routines</td>
<td>116</td>
</tr>
<tr>
<td>5</td>
<td>Arrays</td>
<td>119</td>
</tr>
<tr>
<td>6</td>
<td>Linked Lists</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 8</th>
<th>FUTURE WORK</th>
<th>121</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Chapter 9</th>
<th>CONCLUSION</th>
<th>123</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Appendix A</th>
<th>Computer Program Verification (A survey)</th>
<th>126</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INTRODUCTION</td>
<td>126</td>
</tr>
<tr>
<td>Topic 1</td>
<td>The Problem</td>
<td>127</td>
</tr>
<tr>
<td>Topic 2</td>
<td>Overview of Approaches</td>
<td>127</td>
</tr>
<tr>
<td>Topic 3</td>
<td>Organization of The Paper</td>
<td>128</td>
</tr>
<tr>
<td>1</td>
<td>The role of program verification</td>
<td>128</td>
</tr>
<tr>
<td>2</td>
<td>A basis for program verification</td>
<td>130</td>
</tr>
<tr>
<td>Topic 1</td>
<td>Before Hoare's axiomatic method</td>
<td>130</td>
</tr>
<tr>
<td>Topic 2</td>
<td>Floyd's basis</td>
<td>131</td>
</tr>
<tr>
<td>Topic 3</td>
<td>Hoare's axiomatic method</td>
<td>132</td>
</tr>
<tr>
<td>Topic 4</td>
<td>A theorem of loop assertions</td>
<td>134</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>Topic 5</td>
<td>Other issues</td>
<td>134</td>
</tr>
<tr>
<td>3</td>
<td>Methods</td>
<td>135</td>
</tr>
<tr>
<td>Topic 1</td>
<td>Intermittent assertions</td>
<td>135</td>
</tr>
<tr>
<td>Topic 2</td>
<td>Strong verification</td>
<td>136</td>
</tr>
<tr>
<td>Topic 3</td>
<td>Structural induction</td>
<td>136</td>
</tr>
<tr>
<td>Topic 4</td>
<td>Subgoal induction</td>
<td>137</td>
</tr>
<tr>
<td>4</td>
<td>Verification of some features</td>
<td>137</td>
</tr>
<tr>
<td>Topic 1</td>
<td>Aliasing</td>
<td>138</td>
</tr>
<tr>
<td>Topic 2</td>
<td>Arrays</td>
<td>138</td>
</tr>
<tr>
<td>Topic 3</td>
<td>Functions</td>
<td>139</td>
</tr>
<tr>
<td>Topic 4</td>
<td>Jumps (goto's)</td>
<td>140</td>
</tr>
<tr>
<td>Topic 5</td>
<td>Multiple assignment</td>
<td>142</td>
</tr>
<tr>
<td>Topic 6</td>
<td>Procedure calls</td>
<td>143</td>
</tr>
<tr>
<td>Topic 7</td>
<td>Recursion</td>
<td>144</td>
</tr>
<tr>
<td>Topic 8</td>
<td>Side effects</td>
<td>144</td>
</tr>
<tr>
<td>5</td>
<td>Automatic verification</td>
<td>146</td>
</tr>
<tr>
<td>Topic 1</td>
<td>Axiomatization of Pascal and Euclid</td>
<td>146</td>
</tr>
<tr>
<td>Topic 2</td>
<td>Lucid</td>
<td>146</td>
</tr>
<tr>
<td>Topic</td>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3</td>
<td>VISTA</td>
<td>147</td>
</tr>
<tr>
<td>4</td>
<td>Loop predicates</td>
<td>148</td>
</tr>
<tr>
<td>5</td>
<td>VCG</td>
<td>149</td>
</tr>
<tr>
<td>6</td>
<td>Miscellaneous concepts</td>
<td>150</td>
</tr>
<tr>
<td>1</td>
<td>Conversion</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>Decomposition Rule</td>
<td>151</td>
</tr>
<tr>
<td>3</td>
<td>Equivalence of programs</td>
<td>151</td>
</tr>
<tr>
<td>4</td>
<td>Incorrectness proving</td>
<td>152</td>
</tr>
<tr>
<td>5</td>
<td>Invariant generating</td>
<td>152</td>
</tr>
<tr>
<td>6</td>
<td>Predicate transformers</td>
<td>152</td>
</tr>
<tr>
<td>7</td>
<td>Relational model</td>
<td>153</td>
</tr>
<tr>
<td>8</td>
<td>Termination</td>
<td>153</td>
</tr>
<tr>
<td>9</td>
<td>Well founded sets</td>
<td>154</td>
</tr>
<tr>
<td>7</td>
<td>Summary of Notation</td>
<td>155</td>
</tr>
<tr>
<td>8</td>
<td>CONCLUDING COMMENTS</td>
<td>156</td>
</tr>
<tr>
<td>9</td>
<td>Review of papers</td>
<td>157</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>173</td>
</tr>
<tr>
<td>Appendix B</td>
<td>VITA AUCTORIS</td>
<td>182</td>
</tr>
</tbody>
</table>
Chapter 1 INTRODUCTION

Section 1 Introduction

Object-oriented software construction is very popular owing to the high software quality and productivity associated with it. Due to the popularity of the paradigm we need methods of verifying the correctness of object-oriented programs. The verification of object-oriented programs is the subject of this thesis.

Many of the components of object-oriented programs are mutually dependent. When a component $A$ is dependent on another component $B$, it suffices to show that the correctness of $A$ logically follows from the correctness of $B$ assuming that component $B$ has already been proven correct. In the verification of object-oriented programs we should try and take advantage of the independence of the different software components.

Section 2 Object-Oriented Programming and Software Quality

The quality of a software component consists of its adaptability, correctness, efficiency, extensibility, portability, and reusability. Object-oriented languages are appropriate for the construction of adaptable, extendable, portable and reusable software components. Efficiency is most easily attained through optimization techniques and good programming practice.
Adaptability, extensibility and reusability is where the inheritance of object-oriented programming is valuable. A class is being treated as generalized software every time another component inherits from it. Instead of writing a software component from scratch it is more productive to use previous components that have been proven correct and adapt them for the new application. Inheritable software is reusable software. Polymorphism provides another advantage since polymorphic software is the most reusable type of software.

In object-oriented programming the internal implementation of a class can be altered without affecting any other part of the program. Clients of a class should not be concerned with the internal implementation of that class.

If there is a single most important factor of software quality it is correctness. The object-oriented approach to software development tends to result in fewer programming errors. Reusability means less software to write and, as a result, fewer errors. The catch to this is that a reusable software component must be correct because the semi-popular saying "Every error you make on a computer is repeated 100 times over" is true when it comes to reusable software.

**Section 3 Verification vs Testing**

The methods of attempting to validate a program fall under two major categories. They are program verification and program testing. In this section we compare these two categories.
Program testing involves the execution of the software in an attempt to reveal programming errors. In some cases there are specifications in the form of assertions to be monitored at run time. For program testing, the computer can do the labor for you. Program testing, however, does not guarantee the correctness of software because it can only reveal the presence of errors and not the absence of errors. Program verification, on the other hand, can reveal the absence of errors.

Program verification involves a logical analysis of the software in an attempt to validate the program. It does not involve the execution of the program. Specifications play a major role in program verification. Formal verification reduces arguments about program behavior to a mechanical application of rules.

In theory there is no generally applicable, completely reliable method for proving that a program is totally correct. Termination of programs is not a Turing computable problem. However, a good verifier employing the best available heuristics is a valuable tool. Software that cannot be proven partially correct should not be released.

Section 4 Overview of thesis

The polymorphism in object-oriented programs is one of the major features addressed in this thesis. In this thesis we develop a polymorphic object-oriented mini-language. In this mini-language an explicit distinction is made between the type of a variable and the type of the object assigned to it.
One of the major goals of this thesis is to develop a method for the verification of classes as independent software components. The philosophy is to be able to apply a class specification to a class regardless of the environment. In many cases, a class can be used in more than one application and still preserve its properties. After a class has been proven correct according to its specification, we can use the class specification as an aid in proving the partial correctness of object-oriented programs that use the given class.

A review of the two major topics of this thesis appear in the next 2 chapters. An entire chapter is dedicated to an informal definition and explanation of object-oriented programming. Another chapter reviews the major definitions and theory with respect to program verification.

A chapter is dedicated to the formal definition of an object-oriented mini-language and a system for verifying programs in this mini-language. The chapter immediately following describes methods for defining class specifications, verifying classes with respect to these specifications and using class specifications in the verification of object-oriented programs.

A chapter is dedicated to the definition of a mini-language and proof system for proving programs with exception handling constructs. Exceptions caused by violation of assertions and exceptions thrown on purpose are covered in the mini-language. A non-deterministic construct is introduced to simulate operations
which are non-deterministic in whether or not they will throw an exception. A request for memory is one example.

The first three chapters are the introductory chapters. Chapter 2 describes object-oriented programming and chapter 3 describes program verification. In chapter 4 an object-oriented mini-language appears along with a proof system designed for proving the partial correctness of programs in this mini-language. Chapter 5 introduces class specifications and advocates their usage. Chapter 6 contains the proof system for programs with exception handlers. Chapter 7 introduces techniques for conversion.
Chapter 2  OBJECT-ORIENTED PROGRAMMING LANGUAGES

Section 1 Introduction

In this chapter we give an informal definition of object-oriented programming and we use Eiffel as the primary model. There are many languages supporting object-oriented programming. Some examples are Eiffel, Smalltalk, and C++. Eiffel can be considered completely object-oriented since every Eiffel module is a class. C++ is a dialect of C that supports object-oriented programming. Smalltalk, an interactive object-oriented programming language, is appropriate for experimenting and designing prototypes.

Object-oriented software design is the construction of software systems as structured collections of abstract data type implementations. These abstract data type implementations are called classes.

Section 2 Objects

An object is an entity with state and behavior. The state of an object consists of its private memory. The behavior of an object is represented by a set of routines. Each routine can change the state of the object and/or send messages to other objects.
During execution, objects are the components of the process that executes the object-oriented program. Every object is an instance of a class which dictates the possible state and behavior of the object.

An object is destroyed automatically when it leaves scope. An object is unlike a node which has to be destroyed by an explicit command. This deallocation of memory for objects is hidden from the programmer in the language of Eiffel. In a C++ program if there is dynamically allocated memory within an object it should be deallocated within the associated destructor function.

Section 3 Classes

Classes are the building blocks of object-oriented programs. A class represents a set of run time objects with common features. Here we show a class called \textit{INT\_PAIR} with two attributes and four routines.

```plaintext
class INT_PAIR export
    first, second, make_first, make_second, add, negate;
feature
    first, second: INTEGER;
    make_first(a: INTEGER) is
        do
            first := a;
        end;
    make_second(a: INTEGER) is
        do
            second := a;
        end;
    add(k: INT_PAIR)
        do
            first := first + k.first;
            second := second + k.second;
        end;
    negate is
        do
            first := 0 - first;
            second := 0 - second;
        end;
end
```
An Eiffel program consists of one or more classes. One of these classes is called the root of the system and this is where execution starts. We can define a create routine for a class. A create routine is applied to every instance of a class upon creation.

In C++ we can define constructor and destructor functions for classes. A constructor function is the equivalent of create in Eiffel. A destructor function is invoked on an object when that object goes out of scope. Normally a destructor function is used to deallocate memory. Eiffel does not need any explicit deallocation of memory and therefore has no destructor functions.

A feature is a routine or an attribute. Every feature of a class will be a feature in every instance of that class. An attribute is a component of the private memory of an object. The current state of any instance of a class is determined by the values of its attributes. A routine will either change the state of the object, return a value, or do both. The language of Eiffel contains a feature clause to hold the features introduced in that class. Here we show the different kinds of features allowed in Eiffel. C++ has similar capabilities.

```eiffel
class AREA_OF_CIRCLE export
  Pi,number_of_tests,print_area,area
feature
  Pi:REAL is 3.14159;  -- A constant.
  number_of_tests:INTEGER;  -- A variable.
  print_area(radius:REAL) is  -- A procedure.
    do
      io.putreal(area(radius));
      io.new_line;
  end;
end;
```
number_of_tests := number_of_tests + 1;
end;
area(radius:REAL):REAL is
  do
    Result := Pi * radius * radius;
  end;
end;

An exported feature of a class is a feature that is available to the clients of that class. A client of a class A is any software component that creates or uses objects of type A.

A once routine will execute its body the first time it is invoked on an object, however, subsequent invocations of the routine on the same object have no effect. The value returned by a once function on subsequent calls will be the same value as on the first call. A once routine can be used for the initialization of variables or for shared data.

Section 4 Inheritance

Inheritance is a very powerful tool for software developers who use object-oriented languages. Whenever a class A "inherits" another class B, all the features of class B are implicitly included in the features of class A. Object-oriented languages typically allow a class to inherit from any number of classes. The features of a class consist of the features introduced in the class together with all inherited features. Here we show a class INT_TRIPLE inheriting from INT_PAIR.
class INT_TRIPLE export
  repeat INT_PAIR.third,make_third,add,negate;
inherit
  INT_PAIR
  redefine add,negate;
feature
  third:INTEGER;
  make_third(a:INTEGER) is
do
  third := a;
  add(k:INT_TRIPLE)
do
  first := first + k.first;
  second := second + k.second;
  third := third + k.third;
end;
negate is
do
  first := 0 - first;
  second := 0 - second;
  third := 0 - third;
end;
end;

**Definition 2.4.1** A class $B$ is a descendent of class $A$, and $A$ an ancestor of $B$, if and only if $B$ is $A$ or $B$ inherits from a descendent of $A$. Class $B$ is a proper descendent of class $A$, and $A$ a proper ancestor of $B$, if and only if $B$ is a descendent of $A$ and $B$ is not $A$.

Inherited routines can be redefined. Redefining means that a new implementation is given for the routine. In *Eiffel* inherited features may be renamed to fit into a new context or to resolve name clashing whenever a class inherits from more than one class.

A deferred class (see below) is a definition of a class with at least one feature having no default implementation. Such a feature is called a deferred feature. To be used a deferred class must be inherited by another class and every deferred
feature must be given an implementation in the new class otherwise the new class becomes a deferred class. We cannot create an instance of a deferred class, however, a variable whose type is a deferred class can be used. See section on polymorphism for more details.

Here we show a deferred class with a routine called start. This procedure repeatedly prompts the user for input then passes that input to a routine called process_input. In a descendent of this class where the routine process_input is given an implementation the routine start will repeatedly call this routine with the input from the keyboard.

defered class INTERACTIVE export  .. GENERAL INTERACTIVE CLASS.
  start,stop,done, input_prompt,process_input
feature
done:BOOLEAN;  .. Indicates if done interactive mode.
start is
  local
    xinput:STRING;
do
  from
    done := false;
  until
    done
  loop
    io.putstring(input_prompt);  .. Display prompt.
    io.readline();  .. Read line of input.
    xinput := io.stdin.string.duplicate();  .. Store in xinput.
    process_input(xinput);  .. Process xinput.
  end;
  end;
stop is  .. stop interactive mode.
do
  done := true;
end;
.. *--------------------------------------------------------------------------
.. defered routines
.. *--------------------------------------------------------------------------
input_prompt:STRING is  .. the main input prompt.
  deferred
end;
process_input(xxx:STRING) is  .. process the input.
  deferred
end;
end;
end;
Section 5 Polymorphism and Dynamic Binding

It is possible for an object of type \( B \) to be assigned to a variable of a different type \( A \). The type \( B \) must conform to type \( A \) for this substitution to be valid (see below). The concept of dynamic binding is an important concept in object-oriented programming. Whenever dynamic binding is enforced, the code to execute is not determined at compile time but at run time. Dynamic binding, in object-oriented programming, means that whenever a routine is invoked on an object, the version of the routine to be executed is the one associated with the type of the object and not necessarily the type of the variable.

Definition 2.5.1 For type \( B \) to conform to type \( A \) assuming that \( A \) and \( B \) are different types:

- The class defining type \( B \) must inherit directly or indirectly from the class defining type \( A \).
- Instantiations for all generic parameters in type name \( B \) must conform to the corresponding generic parameters in type name \( A \). (See section on genericity)
- Inherited routines should not be renamed. Remember that if \( B \) is a descendent of \( A \) then \( B \) inherits routines from \( A \) either directly or indirectly.
- Upon inheritance when redefining a routine the types of the parameters and the return value, if any, should not be changed.
In *smalltalk* the variables have no type associated with them and therefore a *smalltalk* variable can refer to any object. Thus *smalltalk* is weakly typed. Dynamic binding is always enforced in *Eiffel* and *Smalltalk*. In C++, however, dynamic binding is only enforced for functions that are defined as *virtual* functions in their base class.

**Section 6 Generic classes**

A generic class, also called a template class, is a definition of a class with parameters representing arbitrary data types. In order to declare a variable using a generic class all parameters require instantiating to an existing data type. The head of a generic class, in *Eiffel*, will take one of two different forms. A couple of examples are *class LINKED_LIST[X]* and *class SORTED_LIST[Y -> PART_COMPAR]*.

The identifier $X$ is treated, within *LINKED_LIST*, as the name of an existing data type. Within the source code for *LINKED_LIST*, however, $X$ is not known to have any features.

The name $Y$ serves the same purpose in *SORTED_LIST* as $X$ in *LINKED_LIST* except that $Y$ is known within *SORTED_LIST* to have features. The features of it are the features of *PART_COMPAR*. The generic parameter $Y$, however, can only be instantiated to data types conforming to *PART_COMPAR*. 

13
The languages of Eiffel and C++ support generic classes. In the language of C++ they are not called generic classes but template classes. Generic software is not specific to object-oriented programming languages. There are some languages which support generic routines and other kinds of generic packages. Generic classes are valuable in cases where several data structures are identical except for a single type name. Generic classes are highly reusable.

Section 7 A final note

C++ is the most popular of the languages supporting object-oriented programming. The advantage of using C++, over Eiffel, is that the programmer has more control over the efficiency of programs. On the other hand, with Eiffel, the programmer need not worry about many of the low level details. In Eiffel there is no temptation to consider low level detail and programming becomes less error prone. The language of Smalltalk is conventionally very slow and is not appropriate as a production language.
Chapter 3  PROGRAM VERIFICATION

Section 1 Introduction

Program verification is the practice of proving, through logical analysis, that programs serve their intended purpose. The merit of the formal methodology is that it replaces unstructured and informal (and possibly incomplete) arguments about the behavior of computation with a systematic application of formal rules.

This chapter discusses program verification and reviews the definitions and theory associated with this field of study. A mini-language definition and a proof method is reviewed in this chapter. This chapter concludes with a proof of the total correctness of a small algorithm according to a given specification.

Section 2 Defining programs

A programming language can be defined through operational semantics [31]. In two different paradigms the base definition may be different, however, a basis for defining programming languages appears below which easily applies to most languages.

Definition 3.2.1  A configuration consists of the following elements:

1. A state: The state $\sigma$ usually assigns values to program variables. The notation $\sigma[x]$ denotes the value of variable, or expression, $x$ in state $\sigma$. 

15
2. A hidden state: This state assigns values to internal variables (sometimes called local variables) that are needed to control the computation, but are not part of the outcome. Correctness assertions do not relate to the values of such variables.

3. A syntactic continuation: This denotes a program, in a language, which represents the rest of the computation to be performed.

The semantics are specified by a transition relation among configurations, denoted by \( C \rightarrow C' \). A computation consists of a sequence of configurations \( C_0 \rightarrow C_1 \rightarrow \ldots \rightarrow C_i \rightarrow \ldots \). The transition relation is an abstract description of a computation step. The notation \( \rightarrow^* \) represents the reflexive transitive closure of \( \rightarrow \).

For deterministic programs the computation of \( P \) starting in the initial configuration \( C_0 = \langle P, \sigma \rangle \) is denoted by \( \pi(P, \sigma) \). Similarly, for non-deterministic programs, \( \Pi(P, \sigma) \) denotes the set of computations starting with \( C_0 = \langle P, \sigma \rangle \).

If \( \pi(P, \sigma) \) is finite then the expression \( \text{val}(\pi(P, \sigma)) \) denotes the final state of \( P \) starting in state \( \sigma \). If \( \pi(P, \sigma) \) is infinite then \( \text{val}(\pi(P, \sigma)) = \bot \) which is the undefined state.

Every programming language \( PL \) can be associated with a meaning function \( M_{PL} \) with the following definition:

\[
M_{PL}[P](\sigma) = \text{val}(\pi(P, \sigma)) \quad \text{in the deterministic case, and}
\]

\[
M_{PL}[P](\sigma) = \{ \text{val}(\pi(P, \sigma)) | \pi \in \Pi(P, \sigma) \} \quad \text{in the non-deterministic case.}
\]
The variables of a program at some point in time will change. A method is required to express the state resulting from changing the value of one or more variables. The variant of the state \( \sigma \) with respect to \( a \) and \( x \), denoted by \( \sigma[a|x] \), is defined by:

\[
\sigma[a|x][v] = \begin{cases} 
  a & \text{if } v \text{ is } x \\
  \sigma[v] & \text{otherwise}
\end{cases}
\]

The expression \( \text{var}(P) \) is used to denote the set of all variables used in \( P \). The expression \( \text{change}(P) \) denotes the set of all variables that \( P \) possibly modifies. The expression \( S_1;S_2 \) represents the concatenation of \( S_1 \) and \( S_2 \). The symbol \( \lambda \) represents the empty program and satisfies the two predicates \( \langle \lambda; S.\sigma \rangle \rightarrow \langle S.\sigma \rangle \) and \( \langle S; \lambda.\sigma \rangle \rightarrow \langle S.\sigma \rangle \). The following rule should hold for any transition relation: If \( \langle A.\sigma \rangle \rightarrow \langle B.\sigma' \rangle \) is a member of the transition relation and \( C \) is a program then \( \langle A; C.\sigma \rangle \rightarrow \langle B; C.\sigma' \rangle \) is also a member of the transition relation.

**Section 3 Specifications**

A specification of a program is a set of conditions which, if satisfied, qualifies the program as correct according to those conditions. A specification of an algorithm usually includes a precondition \( P \) and a postcondition \( Q \) and may be expressed as \( \langle P, Q \rangle \). In order for an algorithm to satisfy specification \( \langle P, Q \rangle \) the algorithm must complete in a state satisfying \( Q \) whenever it starts in a state satisfying \( P \).
An invariant is another type of specification. It is used for expressing that certain properties of the program state are to be preserved. This is appropriate for cyclic programs. The predicate $P$ being an invariant of $S$ can be expressed as $\{P\}S\{P\}$

**Section 4 Partial and Total Correctness**

A program $S$ is considered *partially correct*, with respect to $\langle P, Q \rangle$, if and only if whenever $S$ starts in state satisfying $P$ and then terminates it will terminate in a state satisfying $Q$. This is denoted by $\{P\}S\{Q\}$. By this definition an algorithm which never terminates for any input is considered partially correct. When proving an algorithm, termination is usually proven separately from partial correctness.

A program $S$ is considered *totally correct*, according to $\langle P, Q \rangle$, if and only if whenever $S$ starts in a state satisfying $P$ it will terminate in a state satisfying $Q$. This is denoted by $\langle P \rangle S \langle Q \rangle$. The goal, in program verification, is to prove total correctness.

**Section 5 Termination**

Termination is conventionally proven using well-founded sets. One method is to define a function $F$ on the state of the computer which returns a value from a well-founded set $W$. Next we prove that the value of the function decreases with every “step” of the computation (A loop iteration is a “step”). This proves
termination of a program since the value of $F$ cannot decrease forever since its value is in a well-founded set. This is true by the definition of well-founded sets.

A more flexible method is to define a parameterized invariant $I(w, \bar{x})$ where $w$ is from a well-founded set $\mathcal{W}$. Here we prove that if the contents of a loop start in a state satisfying $I(w_1, \bar{x}_1)$ and not satisfying the loop termination condition, it will end in a state satisfying $I(w_2, \bar{x}_2)$ where $w_2 < w_1$. This proves termination as well.

**Section 6 Soundness and Completeness**

Soundness is a necessary property of formal methods. An unsound method is of no use in program verification. A given formal method of program verification is considered to be sound if and only if every program identified as correct by the method is indeed correct.

Completeness is a valuable property, when it is achieved, but not a necessary one. A given formal method of program verification is considered complete if and only if every program that is correct according to its specifications can be proven by the formal method. In other words a method is complete if and only if it is powerful enough to verify every correct program.

Relative completeness is a conditional completeness. A formal system being relatively complete implies that if there were to exist a system for expressing every predicate on the state of the computer then the system would be complete.
This implies that the source of incompleteness is the inability to express every predicate on the program state.

Section 7 Induction

One of the main mathematical tools of program verification is induction. The two forms of induction are computational induction and structural induction. Computational induction is induction on the progress of the computation. Structural induction is induction on the structure of the objects on which the program operates.

Conventional mathematical induction states that if a statement \( P_i \) is true for some \( i \) and \( P_n \Rightarrow P_{n+1} \) is true for all \( n \geq i \) then \( P_k \) is true for all \( k \geq i \).

Section 8 The language PLW

The following mini-language definition and proof system can be found in [31]. Also see this book for a proof of the soundness of the system. The mini-language is called PLW and its syntax is defined by the following production rules.

\[
S ::= x := e \mid \text{skip} \mid S; S \mid \text{if } B \text{ then } S \text{ else } S \text{ end} \mid \text{while } B \text{ do } S \text{ end};
\]

Definition 3.8.1 The following is a definition of the semantics of the language PLW. The transition relation \( \rightarrow \) among configurations in PLW satisfies the following rules:

1. \( \langle v := e, \sigma \rangle \rightarrow \langle \lambda, \sigma[e[x]] \rangle \)
2. \( \langle \text{skip}, \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle \)

3. If \( \langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle \) then for all \( T \langle S; T, \sigma \rangle \rightarrow \langle S'; T, \sigma' \rangle \)

4. \( \langle \text{while } B \text{ do } S \text{ end }, \sigma \rangle \rightarrow \langle S; \text{while } B \text{ do } S \text{ end }, \sigma \rangle \) if \( \sigma[B] \) is true
   \( \langle \text{while } B \text{ do } S \text{ end }, \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle \) if \( \sigma[B] \) is false

5. \( \langle \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end }, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle \) if \( \sigma[B] \) is true
   \( \langle \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end }, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle \) if \( \sigma[B] \) is false

The following is a proof system for programs in the language PLW. These are the axioms and deductive rules originally discovered by C. A. R. Hoare in 1969.

The assignment axiom is stated as:

\[ \{ P^x \}_{x := e} \rightarrow \{ P \} \]

where \( P^x \) is obtained from \( P \) by substituting \( e \) for every occurrence of \( x \).

The axiom for skip is

\[ \{ P \}_{\text{skip}} \rightarrow \{ P \} \]

The iteration axiom is stated as:

\[ \{ P, B \}_{\text{while } B \text{ do } S \text{ end}} \rightarrow \{ P \land \neg B \} \]

The decision axiom is stated as:

\[ \{ P, B \}_{\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end}} \rightarrow \{ Q \} \]
The rule of composition is:

\[
\frac{\{P\}S_1\{Q\} \land \{Q\}S_2\{R\}}{\{P\}S_1; S_2\{R\}}
\]

The rule of consequence is:

\[
\frac{(P_1 \Rightarrow P_2) \land \{P_2\}S\{Q_2\} \land (Q_2 \Rightarrow Q_1)}{\{P_1\}S\{Q_1\}}
\]

Section 9 Example

In this section we present a small program, a specification and a formal proof that the program satisfies the specification. The following is an function to find the floor of the square root of an integer.

```python
function int_sqrt(n:INTEGER):INTEGER
    do
        i := 0;
        while ((i+1)^2)<=n do
            i := i + 1;
        end;
        return(i);
    end;
```

We claim that whenever `int_sqrt` starts in a state satisfying \{n \geq 0\} it will terminate in a state satisfying the predicate \{i^2 \leq n \land n < (i+1)^2\}. The intended invariant for the while loop is enclosed in square brackets for clarity.

1. \{\((i + 1)^2 \leq n \land (i + 1 \geq 0)\}\}
   
   \[i := i + 1;\]
   
   \{[i^2 \leq n \land i \geq 0]\}
   
   by the assignment axiom
2. \( \{ [i^2 \leq n \land i \geq 0] \land (i + 1)^2 \leq n \land (i + 1 \geq 0) \} \)

\( i := i + 1; \)

\( \{ [i^2 \leq n \land i \geq 0] \} \)

by rule of consequence

3. \( \{ [i^2 \leq n \land i \geq 0] \land (i + 1)^2 \leq n \} \)

\( i := i + 1; \)

\( \{ [i^2 \leq n \land i \geq 0] \} \)

since \( (i \geq 0 \Rightarrow i + 1 \geq 0) \) and \( ((A \Rightarrow B) \Rightarrow ((A \land B) = A)) \)

4. \( \{ [i^2 \leq n \land i \geq 0] \} \)

while \( ((i + 1)^2) \leq n \) do \( i := i + 1; \) end;

\( \{ [i^2 \leq n \land i \geq 0] \land (i + 1)^2 > n \} \)

by rule of iteration

5. \( \{ [i^2 \leq n \land i \geq 0] \} \)

while \( ((i + 1)^2) \leq n \) do \( i := i + 1; \) end;

\( \{ i^2 \leq n \land n < (i + 1)^2 \} \)

by rule of consequence

6. \( \{ 0^2 \leq n \land 0 \geq 0 \} \)

\( i := 0; \)

\( \{ [i^2 \leq n \land i \geq 0] \} \) by the assignment axiom

7. \( \{ n \geq 0 \} \)

\( i := 0; \)
\[
\{ [i^2 \leq n \land i \geq 0] \}
\]

8. \{ n \geq 0 \}

\[
i := 0; \text{ while } ((i + 1)^2) \leq n \text{ do } i := i + 1; \text{ end;}
\]

\[
\{ i^2 \leq n \land n < (i + 1)^2 \}
\]

by 7,5, rule of composition

For proving termination the function \( F = n - i^2 \) is used with the well-founded set being the set of non-negative integers.

The value of the variable \( i \) increases with every iteration. \( i \geq 0 \) is preserved by every iteration of the loop and is, therefore, true after every computation. \( i^2 \) increases with every iteration. \( n \) does not change, therefore, \( n - i^2 \) decreases with every iteration of the loop. It suffices to prove that the expression \( n - i^2 \) is always a non-negative integer. Since \( n - i^2 \geq 0 \) is part of the loop invariant in the proof of partial correctness, \( n - i^2 \) is a non-negative integer throughout the entire computation. Termination has been proven.
Chapter 4 VERIFICATION OF PROGRAMS
WITH POLYMORPHISM

Section 1 Introduction

In this chapter we introduce an object-oriented mini-language designed for verification and a proof system for verifying programs in this mini-language. Polymorphism and dynamic binding are features of this mini-language. An explicit distinction is made between the type of a variable and the type of the object assigned to it.

It is assumed that any primitive data types are algebraic in that the only way to change the state of a primitive variable is by assigning an expression to it. It is also assumed that there are no side effects in the evaluation of expressions involving primitive variables and constants only. The language is otherwise independent of primitive data types. This means that any primitive data types which conform to the above constraints can be used.

Section 2 The state of an object-oriented program

The state of an object-oriented program consists of the state of each of the variables within that program. The possible states of a primitive variable depend on the data type. An object variable can either be attached to an object or contain the value nil. If a variable is attached to an object then the state of the program
is dependent on the state of the object. The type of the object must conform to the type of the variable but is not necessarily the same.

The state of an object consists of the contents of the private memory used by the object and the type of the object. An object is considered to be in a state of execution whenever a routine is currently invoked on that object.

The state of an object-oriented program, without aliasing, can be stated as an object tree. The top node of this tree represents the state of the object-oriented program. Any variable that is assigned an object will have a branch corresponding to each attribute of that object. If \( x \) is a variable in the top level state and \( y \) is an attribute of \( x \) then \( x.y \) denotes the corresponding branch. Similarly \( x.y.v \), if defined, is a branch of \( x.y \).

In a system with aliasing, the state of an object-oriented program can be modelled as a graph where the objects are vertices and the references are edges. This would also prove to be another form of non-termination namely an object continuously calling a routine on itself indirectly through another object. This can only result when there is a circular chain of objects.

**Section 3 Preliminary**

Object-oriented programs do not conform easily to simple state definitions. There is a type associated with every variable. There is also a type associated
with every object which may be different from the type of the variable in an object-oriented program.

Definition 4.3.1 A state $\sigma$ is modelled as a set of variables, each with

1. A name denoted by $v$.

2. A type denoted by $\sigma[type(v)]$, where $type(v)$ is an expression representing the type of the variable $v$ (not necessarily the type of the value assigned to $v$). $type(v)$ is a separate variable and is not affected by the assignment $v := e$.

3. A value denoted by $\sigma[v]$. The associated value, if it is an object, has a type associated with it, denoted by $v.type$.

The value of $v.type$ must be a descendent of $type(v)$ according to the rules of object-oriented programming for Eiffel. The rule POLY (introduced later in this chapter) assumes that the above rule is obeyed within its precondition. If there is a run-time type error then a precondition of $false$ will occur which means that the proof has failed.

A state is modeled as a set of triples where each triple represents a variable. The statement $\sigma[v] = z$ is equivalent to $\exists T((v, T, z) \in \sigma)$ and the statement $\sigma[type(v)] = T$ is equivalent to $\exists z((v, T, z) \in \sigma)$.

There is a unique value called $nil$ where the type of $nil$ conforms to every data type and therefore $nil$ can be assigned to any variable. This is the value held by every variable before it is explicitly assigned a value.
A simple identifier is a sequence of characters such that each character is either a letter, a digit, or an underscore and the first character is a letter. A complex identifier is a sequence of one or more simple identifiers connected by dots. A complex variable represents a deep branch.

An object can be represented by a set of triples, each of the form (name, type, value). The first component is the name of the field. The second component is the type of the field. The final item represents the value currently assigned to the field. The following is a typical object:

\{(x1.\text{INTEGER.4}), (x2.\text{INTEGER.8})\}

If \(\sigma[x] = \{(x1.\text{INTEGER.4}), (x2.\text{INTEGER.8})\}\) then \(\sigma[x.x1] = 4\), \(\sigma[x.x2] = 8\) and \(\sigma[\text{type}(x.x2)] = \text{INTEGER}\).

**Definition 4.3.2** If \(n\) is a simple identifier and \(x\) is a complex identifier

1. \(\sigma[x.n] = z\) means \((n.\ T.\ z) \in \sigma[x]\) for some type name \(T\).
2. \(\sigma[\text{type}(x.n)] = T\) means \((n.\ T.\ z) \in \sigma[x]\) for some \(z\).
3. \(\sigma[n] = z\) means \((n.\ T.\ z) \in \sigma\) for some type name \(T\).
4. \(\sigma[\text{type}(n)] = T\) means \((n.\ T.\ z) \in \sigma\) for some \(z\).

**Definition 4.3.3** The set of variables in state \(\sigma\) is \(\text{vars}(\sigma) = \{x|(x.\ T.\ n) \in \sigma\}\).

The set of variables accessed by the program segment \(S\) is \(\text{var}(S)\). The set of variables appearing in predicate \(P\) is \(\text{var}(P)\).
**Definition 4.3.4**  \( \sigma[a|x] \) is defined by the following:

1. For simple variable \( x \):

\[
\sigma[a|x] = \{(y, T, v') | (y, T, v) \in \sigma \land v' = ((y \text{ is } x)?a : v)\}
\]

2. For complex identifier \( c.x \) where \( x \) is simple:

\[
\sigma[a|c.:x] = \sigma\left[\begin{cases} (y, T, v') | (y, T, v) \in \sigma \land v' = \begin{cases} a & \text{if } y \text{ is } x \\ v & \text{otherwise} \end{cases} \end{cases}\right]|c:]
\]

For the purpose of substitution whenever an expression \( x \) would otherwise replace \( y \) in an expression we do not to replace the \( y \) in \( \text{type}(y) \) with \( x \) as part of the same substitution, however, we must replace the \( y \) in \( \text{type}(y,t) \) with \( x \) for any complex identifier \( t \). To see the logic behind this we can view \( \text{type}(y,t) \) as \( y.\text{type}(t) \).

**Definition 4.3.5** Modified Substitution rules.

1. (simultaneous substitution: \( (z)^{x_1,...,x_n}_{y_1,...,y_n} \))

The expression \( (z)^{x_1,...,x_n}_{y_1,...,y_n} \) is undefined if there exists an integer \( i \) and complex identifier \( b \) such that \( x_i = z.b \). The expression is also undefined if there exists 2 integers \( i \) and \( j \) and a complex identifier \( b \) such that \( x_i = x_j.b \). We could define substitution under such circumstances however we feel it is not necessary. Under conditions where the substitution is defined we have the following list of rules:

\[
(x_i)^{x_1,...,x_n}_{y_1,...,y_n} = y_i
\]

\[
(\text{type}(x_i))^{x_1,...,x_n}_{y_1,...,y_n} = \text{type}(x_i) \text{ (no substitution)}
\]
(\text{type}(x,t))^{y_1 \ldots y_n}_{y_1 \ldots y_n} = \text{type}(y,t)

(x,t)^{y_1 \ldots y_n} = y, t \text{ where } t \text{ is a complex identifier.}

(a \text{ op } b)^{y_1 \ldots y_n} = (a)^{y_1 \ldots y_n} \text{ op } (b)^{y_1 \ldots y_n} \text{ as usual}

2. (many to single) \(P^{y_1 \ldots y_n}_{y} = P^{x_1 \ldots x_n}_{r_1 \ldots r_n}\) where \(\forall i (c_i = y)\)

3. \(x.*\) denotes the set \(\{x.t | t \text{ is a simple identifier}\}\) (Here we use * as a kind of wild card)

4. \(x.**\) denotes the set \(\{x.t | t \text{ is a complex identifier}\}\)

5. \(~(N_1 : T_1 \ldots N_m : T_m) = N_1 : T_1 \ldots \text{.} N_m : T_m\) (shorthand dot notation)

\textbf{Theorem 4.3.6} The rule \(\sigma[x | y] \models P \equiv \sigma \models P^y\) applies even if \(y\) is a complex identifier. (proof omitted)

\textbf{Section 4 The Mini-language POLYPLW}

The mini-language \textit{POLYPLW} (POLYmorphic-PLW) is for expressing an object-oriented program. If \(K\) is a non-terminal then \(K^*\) denotes zero or more \(K\)'s. The symbol \(P\) denotes a program in this language.

\(P ::= C^* \; R^* \; V \; S\)

\(C ::= \text{class } T \text{ inherit } T^* \text{ export } N^* \text{ feature } V \; R^* \text{ end; }\)

\(\quad | \text{class } T \text{ export } N^* \text{ feature } V \; R^* \text{ end; }\)

(If "\text{inherit } T^*" is missing then "\(T^*\)" is assumed to be an empty list)

\(R ::= \text{proc } N(A) : T \; V \; S \text{ end; } | \text{proc } N(A) \; V \; S \text{ end; }\)

(If "\(\cdot : T\)" is missing then "\(\cdot : \text{nil}\)" is assumed)
\[ A ::= N : T.A \mid N : T \]

\[ V ::= N : T : V \mid \lambda \]

\[ S ::= \text{if } B \text{ then } S \text{ else } S \text{ end} \mid \text{while } B \text{ do } S \text{ end} \mid \text{skip} \mid x := e \]

\[ \mid N(E) \mid y := N(E) \mid x_.N(E) \mid y := x_.N(E) \]

\[ \mid x_.\text{forget} ; \mid y := x ; \mid S ; S \]

\[ E ::= e ; E \mid e \text{ denoting a list of expressions.} \]

\[ \text{N denotes a feature name (which is an identifier)} \]

\[ \text{T denotes a type name (which is an identifier)} \]

For any class \( C \) and any routine \( R \) of class \( C \) the following sets of identifiers must be mutually independent: 1) attributes, or data members, of class \( C \), 2) routine names of class \( C \), 3) parameters of routine \( R \) in class \( C \), and 4) local variables of routine \( R \) in class \( C \).

For \( \text{POLYPLW} \) the assignment \( y := x \) places a copy of object \( x \) into \( y \) and \( x = y \) is the condition deep-equal. Therefore objects are modelled as complex values that can be changed by the various operations defined on them. The simulation of aliased objects is done later.

Functions in classes can have side effects on the objects they operate on. At the same time, only assignment statements involving the function call alone are allowed. Expressions with side effects can be translated into statements in this mini-language.
Within the code of any routine in a class definition, every occurrence of an identifier \( v \) representing a data member or routine in the class is implicitly \textit{current}.\( v \). Also within the code of a routine definition, every occurrence of an identifier \( v \) representing a variable which is local to the routine, is implicitly \textit{local}.\( v \).

**Definition 4.4.1** The following is the transition relation on configurations for the mini-language \textit{POLYPLW}.

1. For \( \langle C, \sigma \rangle \rightarrow \langle \lambda, \sigma \cup \{(T_c, \text{class}, k_0)\} \rangle \) such that

   a. \( k_1 = \{(k, T, v)| t \in T; (k, T, v) \in \sigma[t]\} \)

      (The inherited features)

   b. \( k_2 = \{(x, T_a, \text{nil})| x : T_a \in V \} \cup \{(p, \text{proc}, R)| R \in R^* \text{ and } p \text{ is the name of } R \} \cup \{(\text{type}, \text{name}, T_c)\} \)

   c. \( k_0 = \{(k, T, v) \in k_1| \exists T_2, v_2((k, T_2, v_2) \in k_2)\} \cup k_2 \)

   If "\textit{inherit} \( T_i \)" is missing then it is implicitly present and \( T_i \) is the empty set.

2. \( \langle R, \sigma \rangle \rightarrow \langle \lambda, \sigma \cup \{(N, \text{proc}, R)\} \rangle \) where \( R = \text{proc} \ N \ldots \)

3. \( \langle N : T, \sigma \rangle \rightarrow \langle \lambda, \sigma \cup \{(N, T, \text{nil})\} \rangle \) introducing a new variable to the program state.
4. \( (\text{while } B \text{ do } S \text{ end}, \sigma) \rightarrow (\text{while } B \text{ do } S \text{ end}, \sigma) \) if \( \sigma[B] \) is true

\( (\text{while } B \text{ do } S \text{ end}, \sigma) \rightarrow (\lambda, \sigma) \) if \( \sigma[B] \) is false

5. \( (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end}, \sigma) \rightarrow (S_1, \sigma) \) if \( \sigma[B] \) is true

\( (\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end}, \sigma) \rightarrow (S_2, \sigma) \) if \( \sigma[B] \) is false

6. \( (x := e, \sigma) \rightarrow (\lambda, \sigma[e/x]) \) for all expressions \( e \) and variables \( x \).

\( (y := x, \sigma) \rightarrow (\lambda, \sigma[x/y]) \) for object variables \( x \) and \( y \).

(Here we use the extended definition of \( \sigma[e/x] \)).

7. \( (\text{skip}, \sigma) \rightarrow (\lambda, \sigma) \)

8. \( (x.\text{create}(E), \sigma) \rightarrow \langle S^\text{current,local}_{x,ls}, \text{remove}(ls), \sigma[T|x] \cup p \rangle \) where

   a. \( p = \{(ls, \text{localstate}, q)\} \)

   \[ q = \{(v_i, T_i, \sigma[e_i]) | i = 1..n\} \cup \{(N, Z, \text{nil}) | (N : Z) \in V\} \cup \{(\text{Result}, T_r, \text{nil})\} \]

   where \( E = e_1...e_n \) and \( A = v_1 : T_1...v_n : T_n \) and \( ls \) is the unique local object created for the routine call.

   b. \( \sigma[T.\text{create}] = \text{proc create}(A) : T_r V S \text{ end}; \)

   c. \( \sigma[\text{type}(x)] = T \)

9. \( (x.\text{forget}, \sigma) \rightarrow (\lambda, \sigma[\text{nil}[x]]) \)

10. \( (x.N(E), \sigma) \rightarrow \langle S^\text{current,local}_{x,ls}, \text{remove}(ls), \sigma \cup p \rangle \) where

    a. \( p = \{(ls, \text{localstate}, q)\} \)

    \[ q = \{(v_i, T_i, \sigma[e_i]) | i = 1..n\} \cup \{(N, Z, \text{nil}) | (N : Z) \in V\} \cup \]
\{(Result, T_r, nil)\}

where \(E = e_1...e_n\) and \(A = v_1 : T_1...v_n : T_n\) and \(ls\) is the unique local object created for the routine call.

b. \(\sigma[T.N] = proc\ N(A) : T_r \ V\ S\ end;\)

c. \(\sigma[x.type] = T\)

11. \(\langle y := x.N(E), \sigma\rangle \rightarrow \langle S^\text{current,local}_s, y := ls.\text{Result}; \text{remove}(ls), \sigma \cup p\rangle\)

where

a. \(p = \{(ls.\text{localstate}, q)\}\)

\[q = \{(v_i, T_i, \sigma[e_i])|i = 1...n\} \cup \{(N, Z, \text{nil})|(N : Z) \in V\} \cup \{(\text{Result, T_r, nil})\}\]

where \(E = e_1...e_n\) and \(A = v_1 : T_1...v_n : T_n\) and \(ls\) is the unique local object created for the routine call.

b. \(\sigma[T.N] = proc\ N(A) : T_r \ V\ S\ end;\)

c. \(\sigma[x.type] = T\)

12. \(\langle N(E), \sigma\rangle \rightarrow \langle S^\text{local}_s, \text{remove}(ls), \sigma \cup p\rangle\) where

a. \(p = \{(ls.\text{localstate}, q)\}\)

\[q = \{(v_i, T_i, \sigma[e_i])|i = 1...n\} \cup \{(N, Z, \text{nil})|(N : Z) \in V\} \cup \{(\text{Result, T_r, nil})\}\]

where \(E = e_1...e_n\) and \(A = v_1 : T_1...v_n : T_n\) and \(ls\) is the unique local object created for the routine call.
b. \( \sigma[N] = \text{proc } N(A) : T_r V S \text{ end; } \)

13. \( \langle y := N(E), \sigma \rangle \rightarrow \langle S'_{\text{local}} : y := \text{ls.Result:remove(ls), } \sigma \cup p \rangle \) where

a. \( p = \{(\text{ls.localstate, } q)\} \)

\( q = \{(v_i, T_i, \sigma[e_i]) | i = 1..n\} \cup \{(N, Z, \text{nil}) | (N : Z) \in V\} \cup \{(\text{Result, } T_r, \text{nil})\} \)

where \( E = e_1...e_n \) and \( A = v_1 : T_1...v_n : T_n \) and \( \text{ls} \) is the unique local object created for the routine call.

b. \( \sigma[N] = \text{proc } N(A) : T_r V S \text{ end; } \)

14. \( \langle \text{remove(ls), } \sigma \rangle \rightarrow \langle \lambda. \{(x, T.u) \in \sigma | x \text{ is not } \text{ls}\} \rangle \) (this is an auxiliary command)

15. If \( \langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle \) then for all \( T \langle S : T. \sigma \rangle \rightarrow \langle S' : T. \sigma' \rangle \)

**Section 5 Proof System for POLYPLW**

**Definition 4.5.1** Let \( \text{isvar}(x, A) = \forall((N : T) \in A)(\text{type}(x, N) = T) \)

We state an axiom for an object function call and an ordinary function call. To get axioms for the corresponding routine calls that do not return a value simply look at \( y \) as a dummy variable that is never used and thus \( Q_{\text{ls.Result}}^y = Q \). We now state the axioms and deductive rules of the proof system.
(CREATE): For \( T :: proc create(a_1 : T_1 \ldots a_n : T_n) \ V \ S \ end; \) and the set of attributes \( V_r \) from the class definition and all inherited classes we have

\[
\left\{ P \land isvar(ls, V_l) \land isvar(x, V_r) \right\} S^\text{current,local}_x \downarrow ls \{ Q \} \]

\[
\left\{ \left( \left( \left( P^{x, \text{type}} \right)^{x, \ast} \right)^{ls, a_1 \ldots ls, a_n} \right)^{ls, \ast} \right\}^{ls, \ast}_n \land \text{type}(x) = T \}^{x, \text{create}(c_1 \ldots c_n)} \{ Q \}
\]

for a new identifier \( ls \) where \( ls \in \text{var}(Q) \)

Option: Let \( Q = (Q' \land (x, \text{type} = T)) \)

(CREATE.NOPROC) If class \( T \) has no explicit create procedure and \( V_r \) is the set of attributes from the class definition then

\[
\left\{ \left( \left( \left( P^{x, \text{type}} \right)^{x, \ast} \right)^{ls, a_1 \ldots ls, a_n} \right)^{ls, \ast} \right\}^{ls, \ast}_n \land \text{type}(x) = T \}^{x, \text{create}()} \{ P \land isvar(x, V_r) \}
\]

(FORGET):

\[
\{ P^n \}^{x, \text{forget}} \{ P \}
\]

(OBJ-FUNC):

For \( T :: proc N(a_1 : T_1 \ldots a_n : T_n) : T_r \ V \ S \ end; \) and a new identifier \( ls \) we have:

\[
\left\{ P \land isvar(ls, V) \right\} S^\text{current,local}_x \downarrow ls \{ Q^y_{ls, \text{Result}} \}
\]

\[
\left\{ \left( \left( P^{ls, a_1 \ldots ls, a_n} \right)^{ls, \ast} \right)^{ls, \ast}_n \land (x, \text{type} = T) \} \right\}^{y := x.\text{N}(c_1 \ldots c_n)} \{ Q \}
\]

for a new identifier \( ls \) where \( ls \in \text{var}(Q) \)

Option: Use \( \{ Q^y_{ls, \text{Result}} \land (x, \text{type} = T) \} \) in place of \( \{ Q^y_{ls, \text{Result}} \} \) (Leave \( Q \) as is)
(POLY): For a set of versions of the same routine

\[
\{ T_i :: \text{proc } N(a_1 : U_1 \ldots a_n : U_n) : T_r \ V_i \ S_i \ end ; i = 1..m \}
\]

from a set of decedents \( \{ T_i | i = 1..m \} \) of a common class and for a new identifier \( ls \) we have the following rule:

\[
\forall i \left( \{ P_i \land isvar(ls, V_i) \} (S_i)_x^{\text{current, local}} \{ Q_{ls, \text{Result}}^y \} \right)

\left( \bigvee_{i=1}^{m} \left( \left( (P_i)_{ls,a_1 \ldots ls,a_n}^{ls, \ast} \land (x, \text{type} = T_i) \right) \right) \right) y := x.N(e_1 \ldots e_n) \{ Q \}
\]

for a new identifier \( ls \) where \( ls \in \text{var}(Q) \)

Option: Use \( \{ Q_{ls, \text{Result}}^y \land (x, \text{type} = T_i) \} \) in place of \( \{ Q_{ls, \text{Result}}^y \} \)

(COPY): For object variables \( x \) and \( y \)

\[
\{ P_x^y \} y := x ; \{ Q \}
\]

(FUNC): For \( \text{proc } N(a_1 : T_1 \ldots a_n : T_n) : T_r \ V \ S \ end ; \)

\[
\left\{ P \land isvar(ls, V) \right\} S_{ls}^{local} \{ Q_{\text{Result}}^y \}

\left( \left( \left( P_{ls,a_1 \ldots ls,a_n}^{ls, \ast} \right)_{nil} \right) \right) y := N(e_1 \ldots e_n) \{ Q \}
\]

for a new identifier \( ls \) where \( ls \in \text{var}(Q) \)

(ASSN):

\[
\{ P^x_e \} x := e ; \{ P \}
\]

(SKIP):

\[
\{ P \} \text{skip} \{ P \}
\]
(WHILE):

\[ \frac{\{ P \land B \} S \{ P \}}{\{ P \text{ while } B \text{ do } S \text{ end} \} : \{ P \land \neg B \}} \]

(IF):

\[ \frac{\{ P \land B \} S_1 \{ Q \} \land \{ P \land \neg B \} S_2 \{ Q \}}{\{ P \} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end} : \{ Q \}} \]

(COMP):

\[ \frac{\{ P \} S_1 \{ Q \} \land \{ Q \} S_2 \{ R \}}{\{ P \} S_1 ; S_2 \{ R \}} \]

(CONS):

\[ \frac{(P_1 \Rightarrow P_2) \land \{ P_2 \} S \{ Q_2 \} \land (Q_2 \Rightarrow Q_1)}{\{ P_1 \} S \{ Q_1 \}} \]

For recursion and mutual recursion it is necessary to prove the code of each routine based on the assumption that all routine calls are correct. This is a form of induction on the depth of recursion. Note that an appropriate precondition \( P \) and postcondition \( Q \), corresponding to each routine, must be stated.

**Section 6 Examples**

Here is an example with a class called COIN. There are four classes inheriting from COIN. This example shows polymorphism where a variable of type COIN can contain an object of type PENNY, NICKEL, DIME, or QUARTER. It can also be assigned an object of its own type COIN.

```plaintext
class COIN export
  value
feature
  proc value():REAL
    Result:=0;
  end;
end;
```
classes PENNY export
  value
  inherit
  COIN
feature
  proc value():REAL
    Result:=0.01;
  end;
end;
classes NICKEL export
  value
  inherit
  COIN
feature
  proc value():REAL
    Result:=0.05;
  end;
end;
classes DIMES export
  value
  inherit
  COIN
feature
  proc value():REAL
    Result:=0.10;
  end;
end;
classes QUARTER export
  value
  inherit
  COIN
feature
  proc value():REAL
    Result:=0.25;
  end;
end;
c.l:PENNY;
x:COIN;
y:INTEGER;
c.l.create();
x:=c.l;
y:=x.value();

Using OBJ-FUNC we can prove that the above program ends with \( y=0.01 \).

In lines 1 and 2 we look ahead to see the type of the object assigned to \( x \). Had we said \( x\text{.type}=\text{COIN} \) a precondition of \( false \) would have resulted.

1. \{true\}
ls.Result:=0.01;  

{ls.Result=0.01} (ASSN)(ls is the local object)

2. {x.type=PENNY}  
   y:=x.value()  
   {y=0.01} (OBJ-FUNC),l,simplification

3. {c1.type=PENNY}  
   x:=c1  
   {x.type=PENNY} (COPY)

4. {type(c1)=PENNY}  
   c1.create()  
   {c1.type=PENNY} (CREATE.NOPROC)

5. {type(c1)=PENNY}  
   c1.create();  
   x:=c1;  
   y:=x.value()  
   {y=0.01} (2,3,4,CONS applied twice over)

Let us use POLY this time in order to prove a slightly different program. The classes stay the same but the code changes. Here we assume that z is a boolean variable and price is another integer in the program and give the following as the new code:
Here we give an arbitrary postcondition \( y \geq \text{price} \). Let \( ls \) be the "local object" of \( y := x.\text{value}() \). Also let \( Q(t) \) be the long condition:

\[
(t = \text{COIN} \text{ and } 0 \geq \text{price}) \text{ or } \\
(t = \text{PENNY} \text{ and } 0.01 \geq \text{price}) \text{ or } \\
(t = \text{NICKEL} \text{ and } 0.05 \geq \text{price}) \text{ or } \\
(t = \text{DIME} \text{ and } 0.10 \geq \text{price}) \text{ or } \\
(t = \text{QUARTER} \text{ and } 0.25 \geq \text{price})
\]

1. For \( W = 0 \), \( W = 0.01 \), \( W = 0.05 \), \( W = 0.10 \), and \( W = 0.25 \) we have

\[
\{W \geq \text{price}\} \\
\text{ls.Result := W} \\
\{\text{ls.Result} \geq \text{price}\} \text{ (ASSN)}
\]

2. \( \{Q(x.\text{type})\} \\
\quad y := x.\text{value}() \\
\quad \{y \geq \text{price}\} \text{ (POLY) and 1} \]

41
3. \{Q(c2.type)\}
   \hspace{1em}x := c2
   \{Q(x.type)\} (COPY)

4. \{Q(type(c2)) and type(c2) = NICKEL\}
   \hspace{1em}c2.create()
   \{Q(c2.type)\} (CREATE.NOPROC)

5. \{type(c2) = NICKEL and 0.05 >= price and not(z)\}
   \hspace{1em}c2.create();
   \hspace{1em}x := c2
   \{Q(x.type)\} (simplified precondition and CONS)

6. Similarly we can prove
   \{type(c1) = PENNY and 0.01 >= price and z\}
   \hspace{1em}c1.create();
   \hspace{1em}x := c1
   \{Q(x.type)\}

7. \{(z and type(c1) = PENNY and 0.01 >= price) or
   \hspace{1em}(not(z) and type(c2) = NICKEL and 0.05 >= price)\}
   \hspace{1em}if z then
   \hspace{1em}c1.create();
\[ x := c_1 \]

\[
\text{else}
\]

\[ c_2.\text{create}() ; \]

\[ x := c_2 \]

\text{end}

\{ Q(x.\text{type}) \} \text{ (IF)}

8. Since type(c_1)=PENNY and type(c_2)=NICKEL we have:

\{ (z \text{ and } 0.01 \geq \text{price}) \text{ or } (\text{not}(z) \text{ and } 0.05 \geq \text{price}) \}

if \ z \ then

\[ c_1.\text{create}() ; \]

\[ x := c_1 \]

\text{else}

\[ c_2.\text{create}() ; \]

\[ x := c_2 \]

\text{end;}

\[ y := x.\text{value}() \]

\{ y > \text{price} \} \text{ (2,7,COMP)}

\textbf{Section 7 Termination}

For each of the rules for partial correctness, except for \textit{WHILE}, new rules can be obtained by replacing every occurrence of \{ and \} with \{ and \} respectively.
There are several sources of nontermination in an object-oriented program. The following are the sources: 1) Infinitely many iterations of a while loop. 2) An infinitely descending chain of recursive calls which could result from direct or mutual recursion on the same object, an infinitely descending chain of created objects or an object calling a routine on itself through a call to another object (this is only possible when there is object aliasing).

If there is no recursion within an object-oriented program then the only source of non-termination is a while loop executing infinitely many times. If we replaced every occurrence of \{ and \} with ⟨ and ⟩ in the iteration axiom we would be saying that the termination of each iteration implies the termination of the entire while loop. A counter example to this is \texttt{while } i:=i+1; \texttt{end:}. The following is an axiom for termination of a while loop based on well-founded sets:

(\textbf{WHILE-TERM})

\[
\forall n \exists m (\langle P_n \land B \rangle S \langle P_m \rangle \land m < n) \\
\langle \exists y(P_y) \rangle \texttt{while } B \texttt{ do } S \texttt{ end; } \langle \exists z(P_z) \rangle
\]

for a set of predicates \{\(P_w \mid w \in W\}\} where \(W\) is a well-founded set.

We can prove termination of a set of mutually recursive functions by associating a variant expression with every routine. The result of the invariant expression must be from a well-founded set. Also it must be the same well-founded set for every routine in an object-oriented program. Let us consider the following class with \(W\) being any well founded set:
class VARIANT export
  setValue, success, value
feature
  value: W;
  success: BOOLEAN;
create(v: W)
  value := v;
  success := true;
end;
proc setValue(v: W)
  if v = value and success then
    value := v;
    close
    success := false;
  end;
end;
end;

The above is a package containing a variant variable and a boolean value which indicates if the variant is satisfied. The most important property of this class is that you can only call the routine \textit{setValue} on an object of type VARIANT a finite number of times before the \textit{success} attribute is set to \textit{false} indicating a failure to satisfy the variant.

We can include the extra global variable \textit{superVariant} of type \textit{VARIANT} into a program for which we wish to prove termination. At the beginning of the program create the variable \textit{superVariant} with a given value from the set \textit{W}. Do not recreate this variable again. At the same time change every routine \textit{proc routine(A)} \textit{V; S; end;} into the following:

\begin{verbatim}
proc routine(A)
  v;
  superVariant.setValue(some_expression);
  if superVariant.success then
    S
  end;
end;
\end{verbatim}
The second thing that happens is that if the translated program terminates in a state where `superVariant.success=true` then the original program necessarily terminates. Remember that you are proving termination now and if there are any while loops then you must use this method in conjunction with the iteration rule for total correctness, unless you use a local object of type `VARIANT`.

Also note that if there is more than one routine call in the same procedure then we can do the following with every routine call assuming that `temp` is a local variable of type `VARIANT`:

```plaintext
temp:=superVariant;
routine_call;
if superVariant.success then
    superVariant:=temp;
end;
```

What this does is save a copy of the variant so that the value is reset back to the original value at the beginning of the routine call. This makes the method more powerful since we basically decrease the variant upon descending and restore the previous value upon ascension. We thus have an implicit stack. Note that even with restoring the invariant we can still prove that there is no infinitely descending chain of recursive calls.

**Section 8 Soundness of Proof System for POLYPLW**

In this section the soundness of the system is proven. In doing so the following definition of partial correctness is used:
Definition 4.8.1 \( \{P\} S\{Q\} \equiv \forall a, a' \left[ \left( \sigma \models P \land \langle S, \sigma \rangle \leadsto \langle \lambda, \sigma' \rangle \right) \Rightarrow \sigma' \models Q \right] \)

The axioms of PLW from the previous chapter have been inherited by the mini-language in this chapter. The semantics of the inherited programming language constructs are identical and the associated axioms have already been proven. The other axioms need proving.

(CREATE) Performing a sequence of substitutions captures the effect of doing the corresponding sequence of assignments in reverse. To be more precise we have \( \sigma \models (P^n_d)_d \) if and only if \( \sigma[d/v][b/a] \models P \). Note that \( \sigma[nil|ls.s][ls.t] = nil \) for all simple identifiers \( t \).

1. Let \( \sigma \models \left( \left( \left( P^x_{t, \text{type}(x)} \right)_{x, a_1 \ldots x, a_n} \right)_{\text{ls.s}} \right)_{\text{nil}} \land \text{type}(x) = T \)

2. \( \sigma \models \left( \left( P^x_{t, \text{type}(x)} \right)_{\text{nil}} \land \text{type}(x) = T \right)_{x, a_1 \ldots x, a_n} \) \{ls.l|\exists T(t : T \in V_t)\}

since \( P \) should never contain any expression \( x.t \) where \( t \) is not in \( V_c \) or any \( ls.l \) where \( l \) is not in \( V_l \). Also \( \text{type}(x) = T \) is not affected by the substitution.

3. \( \sigma \cup p \models \left( \left( P^x_{t, \text{type}(x)} \right)_{\text{nil}} \right) \land \text{type}(x) = T \land \text{isvar}(ls, V_l) \)

since including the variables of \( p \) effectively assigns a value of \( nil \) to \( ls.l \) then assigns the local variables as usual. We can include the “isvar” predicate because it is modelled by state \( p \).

4. \( (\sigma \cup p)[T|x] \models P \land \text{type}(x) = T \land \text{isvar}(ls, V_l) \land \text{isvar}(x, V_c) \)

since the assignment \( x := T \) will cause \( x.\text{type} \) to be assigned \( \text{type}(x) \) and \( x.t \) to be assigned \( nil \) for all \( \{t|\exists T(t : T \in V_c)\} \).
5. \((\sigma \cup p)[T|x] \models P \land isvar(ls, V_l) \land isvar(x, V_r)\)

6. \(\sigma[T|x] \cup p \models P \land isvar(ls, V_l) \land isvar(x, V_r)\) since \(x\) is not in \(p\).

7. \(\langle x.create(E), \sigma \rangle \rightarrow \langle S_{x,ls}^{current, local}, remove(ls), \sigma[T|x] \cup p \rangle\) from configuration.

8. \(\langle S_{x,ls}^{current, local}, remove(ls), \sigma[T|x] \cup p \rangle \xrightarrow{*} \langle remove(ls), \sigma_1 \rangle\) where \(\sigma_1 \models Q\).

This is from the configuration.

9. We have \(\langle remove(ls), \sigma_1 \rangle \rightarrow \langle \lambda, \sigma_2 \rangle\) for \(\sigma_2 \models Q\) since \(Q\) should not depend on local variables.

10. By definition of partial correctness the axiom has been proven.

(CREATE-NOPROC) This is easily derivable from CREATE by assuming that there does exist a create procedure but that the empty program \(\lambda\) is the code contained within the procedure.

(FORGET) This is simply the assignment \(x := nil;\) for object \(x\).

(OBJ-FUNC) This is similar to the proof of CREATE, however, there is the extra element of a returned value.

1. Let \(\sigma \models (P_{e_1 ... e_n}^{ls,a_1 ... a_n})_{nil} \land x.type = T\)

2. \(\sigma \models (P \land x.type = T)_{e_1 ... e_n}^{ls,a_1 ... a_n}_{nil}\)

   since \(P\) should never contain any expression with \(ls.l\) where \(l\) is not in \(V_l\).

   Also \(x.type = T\) is not affected by the substitution.
3. \( \alpha \cup p \models P \land x.\text{type} = T \land \text{isvar}(ls, V_l) \)

since including \( p \) effectively assigns a value of \( \text{nil} \) to \( ls.l \) then assigns the local variables as usual. We can include the "isvar" predicate because it is modelled by state \( p \).

4. \( \alpha \cup p \models P \land \text{isvar}(ls, V_l) \)

5. \( \langle y := x.\text{N}(E), \sigma \rangle \rightarrow \langle S_{x,ls}^{\text{current local}}, y := ls.\text{Result}; \text{remove}(ls), \sigma \cup p \rangle \)

from configuration.

6. \( \langle S_{x,ls}^{\text{current local}}, y := ls.\text{Result}; \text{remove}(ls), \sigma \cup p \rangle \rightarrow \langle y := ls.\text{Result}; \text{remove}(ls), \sigma_1 \rangle \rightarrow \langle \text{remove}(ls), \sigma_2 \rangle \)

where \( \sigma_1 \models Q_{ls.\text{Result}}^y \) and \( \sigma_2 \models Q \). This is from the configuration.

7. We have \( \langle \text{remove}(ls), \sigma_2 \rangle \rightarrow \langle \lambda, \sigma_3 \rangle \) for \( \sigma_3 \models Q \) since \( Q \) should not depend on local variables.

8. By definition of partial correctness the deductive rule has been proven.

(POLY) Suppose for a set of types \( \{T_i | i = 1..m\} \) and a set of versions of the same routine

\[ \{T_i :: \text{proc} \ N(a_1 : U_1 \ldots a_n : U_n) : T_r \ V_i \ S_i \ \text{end} ; | i = 1..m\} \]

we have

\[ \{P_i \land \text{isvar}(ls, V_i)\} \langle S_i \rangle_{x,ls}^{\text{current local}} \{Q_{\text{Result}}^y\} \]
Then for all $i$ we have
\[
\left\{ \left( (P_i)^{ls_{a_1 \ldots a_n}} \right)_{i \in I} \right\}^{ls_{*}}_{i \in I} \land (x.type = T_i) \bigg) \right\} \{ y := x.N(e_1 \ldots e_n) \} \{ Q \}
\]
by the rule OBJ-FUNC. Here we depend on the following rule which we hope needs no proof.
\[
\frac{\{ P_1 \} S\{ Q \} \land \{ P_2 \} S\{ Q \} \land \ldots \land \{ P_n \} S\{ Q \}}{\{ P_1 \lor P_2 \lor \ldots \lor P_n \} S\{ Q \}}
\]
From this we get
\[
\left\{ \bigvee_{i=1}^{m} \left( \left( (P_i)^{ls_{a_1 \ldots a_n}} \right)_{i \in I} \right) \land (x.type = T_i) \bigg) \right\} \{ y := x.N(e_1 \ldots e_n) \} \{ Q \}
\]
which is the intended consequent. The deductive rule has been proven.

(FUNC) This is similar to the proof of (OBJ-FUNC) and is therefore omitted.

(COPY) See (ASSN)

(ASSN) from PLW.

(SKIP) from PLW

(WHILE) from PLW

(IF) from PLW

(COMP) from PLW

(CONS) from PLW

50
Section 9 Aliased Objects

This section will be brief since it is not the main point of this thesis. In this section we discuss the fact that certain objects can have more than one reference to them and thus they are aliased objects. An aliased object is an object for which there is more than one method of referring to it. If variables $x$ and $y$ are aliases of each other than the assignment $x := x + 1$ has the effect of adding one to the variable $y$. Here we simulate the aliasing of objects through pointer semantics. Objects can have pointers to them. We do not consider pointers to primitive values as it can easily be simulated. Let us look at the following code:

```plaintext
class Q export
  increment, getval
feature
  value: INTEGER;
  proc create(w: INTEGER)
    value := w;
  end;
  proc increment(w: INTEGER)
    value := value + w;
  end;
  proc getval(): INTEGER
    Result := value;
  end;
end;

x, y: Q;
x.create(7);
a1 := x.getval();
y := x;
y.increment(1);
a2 := x.getval();
```

In this example if $y := x$; is an object copying operation then the operation following does not affect $x$ and $a2$ will be 7. If, however, it is a reference copying then $y.increment(1)$; affects $x$ since $x$ and $y$ are referring to the same
object. In the latter case $a2$ would be 8.

Here we model the object separately from the variable. Separate names are given for each. Upon creation of an object an arbitrary variable name must be introduced which represents the new object. Here is a new addition to the transition relation for the mini-language to allow for object pointers. We now have to write $y:T\&$ to declare a pointer to an object because we want to differentiate it from $y:T$ which is not a pointer to an object but simply an object variable. The value of the expression $&d$, a pointer to object $&d$, is constant throughout the entire life of the variable $d$ which is the entire execution of the program.

1. \( \langle x.create(E), \sigma \rangle \rightarrow \left\langle S^{current,local}_{d,ls}; remove(ls), \sigma[&ns|x] \cup p \cup nil \right\rangle \) where

   a. $d$ is the name given to the newly created dynamic object.

   b. $p = \{(ls, localstate, q)\}$

      \[ q = \{(w_i, T_i, \sigma[i]) | i = 1..n\} \cup \{(N, Z, nil) | (N : Z) \in V\} \cup \{(Result, T_r, nil)\} \]

      where $E = e_1...e_n$ and $A = v_1 : T_1...v_n : T_n$ and $ls$ is the unique local object created for the routine call.

   c. $nil = \{(d, @dynamic, \sigma[T])\}$

   d. $R = \sigma[T.create] = proc create(A) : T_r V^* S end;$

   e. $\sigma[type(x)] = T\&$

2. \( \langle x.N(E), \sigma \rangle \rightarrow \left\langle S^{current,local}_{d,ls}; remove(ls), \sigma + p \right\rangle \) where
a. \( p = \{(ls, localstate, q)\} \)

\[ q = \{(v_i, T_i, \sigma[v_i])|i = 1..n\} \cup \{(N, Z, nil)|\{N : Z\} \in V\} \cup \{(\text{Result}, T_r, \text{nil})\} \]

where \( E = e_1...e_n \) and \( A = v_1 : T_1...v_n : T_n \) and \( ls \) is the unique local object created for the routine call.

b. \( R = \sigma[T.N] = \text{proc } N(A) : T_r \ V \ S \text{ end;} \)

c. \( \sigma[x] = \&d \) and \( \sigma[d.type] = T \)

3. \( \langle y := x.N(E), \sigma \rangle \rightarrow \left\langle S_{d,ls}^{\text{current,local}}, y := ls.\text{Result}, \text{remove}(ls), \sigma + p \right\rangle \)

where

a. \( p = \{(ls, localstate, q)\} \)

\[ q = \{(v_i, T_i, \sigma[v_i])|i = 1..n\} \cup \{(N, Z, nil)|\{N : Z\} \in V\} \cup \{(\text{Result}, T_r, \text{nil})\} \]

where \( E = e_1...e_n \) and \( A = v_1 : T_1...v_n : T_n \) and \( ls \) is the unique local object created for the routine call.

b. \( R = \sigma[T.N] = \text{proc } N(A) : T_r \ V \ S \text{ end;} \)

c. \( \sigma[x] = \&d \) and \( \sigma[d.type] = T \)

4. \( \langle y := x, \sigma \rangle \rightarrow \langle \lambda, \sigma[x[y]] \rangle \) if \( x \) and \( y \) are both reference type and the type of the object pointed to by \( x \) conforms to the type of the variable \( y \).

5. \( \langle y := \text{copy}(x), \sigma \rangle \rightarrow \langle \lambda, \sigma[\&dy[y]] \cup \{(dy, \text{dynamic}, \sigma[dx])\} \rangle \) (where \( x = \&dx \)) for a new name \( dy \) if \( x \) and \( y \) are both reference type and the type
of the object pointed to by $x$ conforms to the type of the variable $y$.

The following axiom is for the routine call on objects by their references.

**(PTR-OBJ-FUNC)** For $T :: proc\ N(a_1 : T_1 \ldots a_n : T_n)\ V_I\ S\ end$; we have

$$\{ P \land isvar(ls, V_I) \} S^{\text{current,local}}_{ls} L_{ls, result} \{ Q^y \}_{ls \text{, result}}$$

$$\left\{ \left( P_x^{lsl_a, \ldots l_s, a_n} \right)^{ls} \land x = &d \land (d.type = T) \right\} y := x.N(v_1 \ldots v_n)\{ Q \}$$

where $ls$ is a newly introduced name.

An axiom similar to POLY can be developed for pointers to objects as well, however, we do not develop such a rule here. The following axiom is for the creation of a new object with a reference pointing to it.

**(PTR-CREATE)** For $T :: proc\ create(a_1 : T_1 \ldots a_n : T_n)\ V_I\ S\ end$; and the set of attributes $V_c$ from the class definition and all inherited classes we have

$$\{ P \land isvar(ls, V_I) \land isvar(d, V_c) \} S^{\text{current,local}}_{ls} \{ Q \}$$

$$\left\{ \left( \left( P_x^{d.type} \right)^{lsl_a, \ldots l_s, a_n} \right)^{ls} \land type(x) = T \right\} x.create(v_1 \ldots v_n)\{ Q \}$$

where $d$ and $ls$ are newly introduced names.

**(PTR-COPY)** for object reference variables $x$ and $y$ we have the following rule for a new variable name $dy$.

$$\{ P^{dy \land a} \land x = &d x \} \{ dy := copy(x) \} \{ P \}$$

The assignment $x := y$ is only valid if both $x$ and $y$ are in the same type category. The three type categories are primitive, object, and pointer to object.
The assignment $y:=\&x$ is not included in this language but can be easily simulated by making $x$ a pointer to an object and using $y:=x$ instead. A pointer variable can only point to dynamically allocated memory. All other capabilities can be simulated in this mini-language.
Chapter 5  VERIFICATION USING CLASS SPECIFICATIONS

Section 1 Introduction

In this chapter we introduce the class specification. A class specification states properties of a class and is a valuable tool in the verification of object-oriented programs. With a specification of its own a class may be verified independently of the rest of an object-oriented program. If this is the case it is easier to reuse the software.

The type of specification dealt with in this chapter will specify both the state and the behavior of instances of a class. The components of the class specification are supported by the language of Eiffel. Eiffel assertions can be monitored at run time, however, there are currently no verification tools for Eiffel.

In this chapter we assume that a class cannot access global variables and that only copies of objects can be passed down or returned from object function calls.

Section 2 Class specifications

This section introduces a definition for a class specification. This definition includes a grammar for the specification. This definition must also indicate what it means for a class to be correct according to a specification.
The definition of a class specification appearing in this chapter assumes that the only routines that will ever be called by another object or the main program are the routines that are specified.

The specification of a class consists of the following:

1. **Memory**: This consists of a set of attributes that are to be part of the class that the specification is applied to. The data types must already exist.

2. **A set of routine specifications**: A routine specification usually consists of a list of arguments to the routine, a precondition, and a postcondition. A precondition is a predicate on the state of the object and any parameters passed down. A postcondition is a predicate on the state of the object, the value returned (assuming the routine is a function), and the final value of any formal parameters (Formal parameters are not considered in this chapter). The state of an object assigns values to the attributes much like the state of a program assigns values to variables.

Below is a suggested grammar for a class specification. This looks a little like a class itself except that \( R \) contains no actual code. Instead it contains a precondition and a postcondition.

\[
Sp ::= \text{class spec } V \_ R^* \text{ end;}
\]

\[
V ::= x : T;
\]

\[
R ::= \text{proc } N(A) : T \ (B_1, B_2)
\]
A specification of the above form is valid if and only if each of the following are true:

1. \( B_1 \) may contain variables found in \( V_c \) and \( A \)
2. \( B_2 \) may contain variables from \( V_c \) and may contain \( Result \).
3. The conditions may contain no other variables.

It would make no sense to have a routine specification which contains reference to local variables and therefore there should be no occurrence of any local variable name within a precondition, postcondition or invariant.

**Definition 5.2.1** \[ isnar(N_1 : T_1; \ldots; N_m : T_m) = \forall(i = 1 \ldots m)(\text{type}(N_i) = T_i) \]

It would make no sense to define a specification without defining what it means for a program to be correct according to a given specification. This definition assumes that the only routines that are called are the routines specified in \( Sp \).

**Definition 5.2.2** A class \( C \) is partially correct according to specification \( Sp = \)

\[ \text{class spec } V_s R^* \text{ inv : } I \text{ end;}, \text{ denoted } C \models Sp, \text{ if and only if:} \]

1. Every attribute in \( V_s \) and routine specification in \( R^* \) is implemented in class \( C \).
2. For every routine \( \text{proc } N(A) : T_r V_r S \text{ end} \) in class \( C \) and every matching in \( \text{proc } N(A) : T_r (P,Q) \) in \( Sp \) the condition \( \{ P \land isnar(V_s; V_r; Result : T_r) \} S\{Q\} \) is true.
Once you have a class and specification that has been proven consistent, or assumed to be consistent, this specification can be used instead of using the routine call axioms in the previous chapter. In using all the axioms in the previous chapter, without some shortcuts, the time for proving programs correct would be proportional to the total number of routine calls made during execution.

**Theorem 5.2.3** Suppose \( C = \text{class } T_r \ldots \) is partially correct with respect to \( S_p = \text{class spec } V_r \ R^* \ \text{inv} : \ I \ \text{end} \). Then if \( \text{proc } N(A) : T_r \ V_r \ S \ \text{end} \) is in \( C \) and \( \text{proc } N(A) : T_r \ (P, Q) \) is in \( S_p \) then for all routine calls

\[
\{ P_{v_1 \ldots v_n, current} \land (x:\text{type} = T_r) \land Z \} y := x.N(E) \{ Q_{y, Result, current} \land Z \}
\]

for the adaptation predicate \( Z \) such that \( x \) does not belong to \( \text{var}(Z) \).

The extra predicate \( Z \) is an adaptation predicate. This was introduced for routine calls in [66]. The predicate \( Z \) can be expanded to include any invariants of the given routine on the object. Note that you can change \( S \), reprove the routine according to the same specifications and any code whose correctness was proved solely from the specification need not be reproved.

**Section 3 Sending Messages to Self**

This algorithm is intended to work in the presence of routine calls that an object makes on itself. The algorithm also works for mutually recursive routines within the same class. This can also be adapted to work for a set of mutually recursive routines between classes.
**Inductive Algorithm:** This algorithm is for proving the partial correctness of the routines in class \( C = \text{class } T_r \ldots \) according to specification \( Sp = \text{class spec } V_r \ R^* \ \text{end}; \)

1. Make sure the partial correctness of any other class used by \( C \) has been proven.

2. Every attribute in \( V^* \) and routine specification in \( R^* \) is implemented in class \( C \).

3. For each routine \( R = \text{proc } N(A) : T_r \ V_r \ S \ \text{end} \) in \( C \) with a routine specification \( \text{proc } N(A) : T_r \ (P.Q) \ \text{in } Sp \) assume that \( \{P^{1:...:n}\} y := N(E)\{Q^\text{Result}\} \). (Note that \( y := N(E) \) is implicitly \( y := \text{current}.N(E) \).)

4. For each routine \( R \) in \( C \) that is specified in \( Sp \) prove \( \{P \land \text{isvar}(V_r; V_r; \text{Result} : T_r)\} S\{Q\} \) solely from the assumptions in step 3.

**Section 4 Class specifications and Inheritance**

Suppose class \( B \) inherits from class \( A \) and class \( A \) is correct according to specification \( Sp \). Can we use \( Sp \) to find a specification for class \( B \)? Before we start, a few definitions are in order.

A routine \( N \) of class \( A \) is said to be *explicitly redefined* in \( B \) if a new implementation has been given. A routine is said to be implicitly redefined if
any routine it calls either directly or indirectly is redefined. A routine is faithfully inherited if it is neither implicitly or explicitly redefined.

The purpose of this next theorem is to show, with fewer lines, that an inherited class satisfies the same specification as the original class. In fact the only routines that have to be reproved are the routines that were explicitly redefined.

**Theorem 5.4.1**  If class \( B \) inherits from class \( A \) and

1. Class \( A \) is proven correct according to specification \( S_p \) by the inductive algorithm.

2. If: For each routine \( R \) in \( B \) and every matching routine specification
   \[
   \text{proc } N(A) : T_r (P, Q) \text{ end: in } S_p \{ P_{e_1, \ldots, e_n y} \} y := N(E) \{ Q_{y}^{\text{Result}} \}
   \]
   Then: For each routine \( R \) that is specified in \( S_p \) and is explicitly redefined in \( B \) we have \( \{ P \land isvar(V_s : V_r : Result : T_r) \} S_\{ Q } \).

then class \( B \) can be proven correct according to specification \( S_p \) by the inductive algorithm.

The above theorem says that the only routines that need reproving are the routines that are explicitly redefined. The routines that are only implicitly redefined have already been proven in the inductive algorithm.

It is not always going to be the case that one specification satisfies a class and all its descendents at the same time. We need to define rules to accom-
modate situations where part of the specification has to be changed in order to accommodate the new situation.

Let us consider the following situation: Class $A$ is a class which satisfies specification $S_I$ and class $B$, which inherits from class $A$, does not satisfy specification $S_I$. We want to create a new specification $S_2$ which is satisfied by class $B$ and is also satisfied by class $A$. Let us further assume that we want the invariant to remain the same in the new specification.

First of all the routine specifications in $S_I$ for each routine that is neither implicitly or explicitly redefined by $B$ can be carried over to $S_2$. Routines that are implicitly redefined may have different specifications. This is because if routine $X$ calls routine $Y$ and routine $Y$ is redefined in class $B$ then routine $X$ may not work the same as in class $A$.

Section 5 Class Invariant

For invariants let us include the syntax $inv: I$ within the class specification and give the following definition.

**Definition 5.5.1** The class specification $class spec\ V_\ast\ R_1\ldots R_n\ inv : I\ end$ is the class specification $class spec\ V_\ast\ (R_1)^I\ldots(R_n)^I\ end$ where $(proc\ N(A) : T_r\ (P,Q))^I = proc\ N(A) : T_r\ (P \land I,Q \land I)$ if $N$ is not create and $(proc\ create(A)\ (P,Q))^I = proc\ create(A)\ (P,Q \land I)$
A class specification now consists of preconditions, postconditions, and an invariant. The above is appropriate under the assumption that a routine will always be called in a state satisfying a precondition and an invariant. Let us assume that such a class specification is consistent with a class. If a routine call is made in a state satisfying the invariant but not satisfying any precondition then we cannot guarantee that the invariant will be preserved.

**Definition 5.5.2** The condition $I$ is a *complete invariant* of class $C$ if and only if for every exported routine $\text{proc } N(A) \text{ } V S \text{ } end$ in class $C$ we have $\{isvar(V)\}S\{I\}$ if $N$ is create and $\{I \land isvar(V)\}S\{I\}$ otherwise.

This definition is interesting because if $I$ is a complete invariant of class $C$ then any object of type $C$ is guaranteed to satisfy the condition $I$ whenever the object is not in a state of execution. This fact can be used in the verification of programs using this class.

**Definition 5.5.3** A specification $Sp$ is a *complete specification* for class $C$ if and only if

1. $C \models Sp$
2. All exported routines in class $C$ are specified in specification $Sp$.
3. There exists a condition $I$ such that
   
   a. For all $\text{proc } N(A) : T (P, Q)$ where $N$ is exported in $C$ we have $Q \Rightarrow I$
b. For all $N \neq \text{create}$ exported in $C$ we have

$$I \Rightarrow \text{or}\{P|\{\text{proc } N(A) : T(\langle P, Q \rangle) \text{ end} \} \in R^*\}$$

c. For the create routine

$$\text{true} \equiv \text{or}\{P|\{\text{proc create}(A) : T(\langle P, Q \rangle) \text{ end} \} \in R^*\}$$

where $\text{or}\{P_1, P_2, \ldots, P_n\} \equiv P_1 \lor P_2 \lor \ldots \lor P_n$

What is the significance of the complete specification? For one thing a complete specification can always be used for every routine call on an object of type $C$. The condition $I$ above is a complete invariant of class $C$ (proof omitted).

**Section 6 Property Dependence**

Sometimes the properties of a class may depend on certain properties being true of objects passed down as parameters to the classes routine members. In a polymorphic environment if you wish to reuse a given class then, normally, you need to know which classes are available which conform to a given data type. To solve this problem we state, within the precondition, that the type of an object passed down (not the type of the variable) satisfies a separate class specification.

We can use the word *models* in place of the symbol $\models$. The following is an example the above situation ($L$ is used as a logical variable):
class spec
  proc foreach(p:INT_LIST) (p.type models Sx, true)
end

where
Sx = class spec
  position: INTEGER;
  count: INTEGER;
  start() (true, position=1)
  forth() (position = 1, count, position=1)
  forth() (count=L, count=L)
  off(): BOOLEAN (position<count, Result=true)
  off(): BOOLEAN (position=count, Result=false)
  off(): BOOLEAN (position=L, position=L)
  off(): BOOLEAN (count=L, count=L)
  item(): INTEGER (position=L, position=L)
  item(): INTEGER (count=L, count=L)
end

Note that it is p.type and not INT_LIST that is used. We do not know the type of the parameter p except that it inherits from INT_LIST. Here is an implementation for the above class specification (Note that the use of p.type models Sx proves termination):

class OPERATOR export
  operation, foreach
feature
  proc foreach(p:INT_LIST)
    k: BOOLEAN;
    kk: INTEGER;
    p.start()
    k:=p.off();
    while not k do
      kk:=p.item()
      operation(kk);
      p.forth();
      k:=p.off();
    end;
  end;
  proc operation(k:INTEGER)
    ...
  end;
end;

Note that the extra class specification Sx guarantees that the routine “foreach” will terminate because the three operations item, forth and off do not change
count and the operations item and off do not change position. The operation forth should increase position by 1 thus bringing it closer to count and thus the routine foreach terminates.

Section 7 Example

Here is a specification and an implementation for a template class called SET.

The following is the class with the type X representing an arbitrary data type.

class SET[X] export
  contains, include, exclude
feature
    a: ARRAY[X];
    c: INTEGER;
    proc create() is
      c:=0;
    end;
    proc contains(v:X): BOOLEAN  iii: INTEGER;
      iii := position(v);
      Result:=true;
    end;
    proc include(v:X)   iii: INTEGER;
      iii := position(v);
      if iii < c then
        c:=c+1;
        a(c):=v;
      else
        skip;
      end;
    end;
    proc exclude(v:X)     iii: INTEGER;
      iii := position(v);
      if iii < c then
        a(iii):=a(c);
        c := c - 1;
      else
        skip;
      end;
    end;
    proc position(v:X):INTEGER
      Result:=1;
      while (Result <= c) and not (v = a(Result)) do
        Result := Result + 1;
      end;
    end;
end;
The specification appears below. The operator "in" implies set inclusion and the operator "?" is the ternary operator of the language c. A couple of definitions are stated here for this example only:

$P_n(A, M, E) = \{n|0 < n \leq M \land A(n) = E\}$ is the set of positions between 1 and $M$ in array $A$ that contain element $E$

$I(A, M) = \forall v(P_n(A, M, v) = \{\} \lor \exists n(P_n(A, M, v) = \{n\}))$ Indicates that array $A$ can only contain one occurrence of any element in the positions 1 to $M$.

class spec

a:ARRAY[X];

c:INTEGER;

proc create()

(true, count = 0)

proc contains(v:X)

(∃n(P_n(a, c, v) = \{n\}), Result)

(P_n(a, c, v) = \{}, ¬Result)

(P_n(a, c, L_1) = L_2, P_n(a, c, L_1) = L_2)

proc include(v:X)

(I(a, c). P_n(a, c, v) = \{L\})

(P_n(a, c, L) = \{\} \land L ≠ v, P_n(a, c, L) = \{\})

(∃n(P_n(a, c, L) = \{n\}) \land L ≠ v, ∃n(P_n(a, c, L) = \{n\}))
proc exclude(v:X)
    \( (\text{true}, Ps(a, c, v) = \{\}) \)
    \( (Ps(a, c, L) = \{\} \land L \neq v, Ps(a, c, L) = \{\}) \)
    \( (\exists n (Ps(a, c, L) = \{n\}) \land L \neq v, \exists n (Ps(a, c, L) = \{n\})) \)

proc position(v:X)
    \( (Ps(a, c, v) = \{L\}, Result = L \land Result \leq c) \)
    \( (Ps(a, c, v) = \{\}, Result > c) \)

Proof of Create:

1. \( \{0 = 0\} \text{ count}=0; \{\text{count} = 0\} \)
2. \( \{\text{true}\} \text{ count}=0; \{\text{count} = 0\} \)

Proof of contains:

Proof for Specification 1:

1. \( \{iii \leq c\} \text{ Result}:=(iii<=c) \{Result\} \)
2. \( \{Ps(a, c, v) = \{L\}\} \text{ iii}=\text{position}(v) \{iii \leq c\} \)
3. \( \{Ps(a, c, v) = \{L\}\} \text{ iii}=\text{position}(v);\text{Result}:=(iii<=c) \{Result\} \)

Proof for Specification 2:

1. \( \{iii > c\} \text{ Result}:=(iii<=c) \{\neg Result\} \)
2. \( \{Ps(a, c, v) = \{\}\} \text{ iii}=\text{position}(v) \{iii > c\} \)
3. \( \{Ps(a, c, v) = \{\}\} \text{ iii}=\text{position}(v);\text{Result}:=(iii<=c) \{\neg Result\} \)

68
Case 3: This should be obvious from the fact that neither \( a \) nor \( c \) is changed by applying this routine.

**Proof of include:**

Proof for Specification 1:

1. \( \{ Ps(a(c,v),c,v) = \{ L \} \} \ a(c) := v \ \{ Ps(a,c,v) = \{ L \} \} \)

   \( \{ Ps(a(c,v),c,v) = \{ c \} \} \ a(c) := v \ \{ Ps(a,c,v) = \{ c \} \} \)

2. \( \{ Ps(a(c+1,v),c+1,v) = \{ c+1 \} \} \ c := c+1; \ \{ Ps(a(c,v),c,v) = \{ c \} \} \)

   \( \{ Ps(a(c+1,v),c+1,v) = \{ c+1 \} \} \ " \ " \)

   \( \{ Ps(a(c+1,v),c,v) = \{ } \wedge a(c+1,v)(c+1) = v \} \ " \ " \)

   \( \{ Ps(a(c+1,v),c,v) = \{ } \} \ " \ " \)

   \( \{ Ps(a,c,v) = \{ } \} \ c := c+1; \ \{ Ps(a(c,v),c,v) = \{ c \} \} \)

3. \( \{ Ps(a,c,v) = \{ } \} \ c := c+1; a(c) := v \ \{ Ps(a,c,v) = \{ c \} \} \) (COMP,1,2)

4. \( \{ Ps(a,c,v) = \{ L \} \} \) skip \( \{ Ps(a,c,v) = \{ L \} \} \)

5. \( \{ ((iii > c) \wedge Ps(a,c,v) = \{ } \) \lor \( ((iii \leq c) \wedge \exists n(Ps(a,c,v) = \{ n \})) \}) \)

   if \( iii > c \) then \( c := c+1; a(c) := v \) else skip end;

   \( \{ \exists n(Ps(a,c,v) = \{ n \}) \} \)

6. \( \{ Ps(a,c,v) = \{ } \} \)

   \( \)iii := position(v)

   \( \{ ((iii > c) \wedge Ps(a,c,v) = \{ } \}

7. \( \{ Ps(a,c,v) = \{ L \} \} \)

69
iii:=position(v)

\{(iii ≤ c) \land (iii = L) \land \exists n(Ps(a, c, v) = \{n\})\}

8. \{Ps(a, c, v) = \{\} \lor \exists n(Ps(a, c, v) = \{n\})\}

iii:=position(v)

\{((iii > c) \land Ps(a, c, v) = \{\}) \lor ((iii ≤ c) \land \exists n(Ps(a, c, v) = n))\} (OR, 7.6)

9. \{Ps(a, c, v) = \{\} \lor \exists n(Ps(a, c, v) = \{n\})\}

iii:=position(v); if iii>c then c:=c+1; a(c):=v else skip end;

\{\exists n(Ps(a, c, v) = \{n\})\} (COMP)

10. \{I(a, c)\}

iii:=position(v); if iii>c then c:=c+1; a(c):=v else skip end;

\{\exists n(Ps(a, c, v) = \{n\})\} (CONS)

Proofs for specifications 2 and 3 are omitted.

Proof of exclude:

Proof for Specification 1:

1. \{Ps(a, c - 1, v) = \{\} \} c:=c-1 \{Ps(a, c, v) = \{\} \}

2. \{Ps(a(iii, a(c)), c - 1, v) = \{\} \} a(iii):=a(c) \{Ps(a, c - 1, v) = \{\} \}

3. \{Ps(a(iii, a(c)), c - 1, v) = \{\} \} a(iii):=a(c); c:=c-1 \{Ps(a, c, v) = \{\} \}

4. \{Ps(a, c, v) = \{\} \} skip \{Ps(a, c, v) = \{\} \}

5. \{(iii ≤ c \land Ps(a(iii, a(c)), c - 1, v) = \{\}) \lor (iii > c \land Ps(a, c, v) = \{\})\}

if iii=c then a(iii):=a(c); c:=c-1 else skip end
\{ P_s(a, c, v) = \{ \} \}

6. \{ \exists n(P_s(a, c, v) = \{ n \}) \}

\text{iii:=position(v)}
\{(\text{iii} \leq c \land P_s(a(iii, a(c)), c - 1, v) = \{ \})\}

7. \{ P_s(a, c, v) = \{ \} \}

\text{iii:=position(v)}
\{(\text{iii} > c \land P_s(a, c, v) = \{ \})\}

8. \{ \exists n(P_s(a, c, v) = \{ n \}) \lor P_s(a, c, v) = \{ \} \}

\text{iii:=position(v)}
\{(\text{iii} \leq c \land P_s(a(iii, a(c)), c - 1, v) = \{ \}) \lor (\text{iii} > c \land P_s(a, c, v) = \{ \})\}

(OR)

9. \{ \exists n(P_s(a, c, v) = \{ n \}) \lor P_s(a, c, v) = \{ \} \}

\text{iii:=position(v); if iii<=c then a(iii):=a(c); c:=c-1 else skip end}
\{ P_s(a, c, v) = \{ \} \} (COMP)

10. \{ I(a, c) \}

\text{iii:=position(v); if iii<=c then a(iii):=a(c); c:=c-1 else skip end}
\{ P_s(a, c, v) = \{ \} \} (COMP)

Proofs for specifications 2 and 3 are omitted.

\textbf{Proof of position:}

Proof for Specification 1:
1. \( \{Ps(a,c,v) = \{L\} \land 1 \leq Result + 1 \leq L\} \)

Result:=Result+1

\( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L\} \) (CONS)

2. \( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L \land Result \neq L\} \)

Result:=Result+1

\( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L\} \) (CONS)(\(L \leq c\) is implicit in the first condition).

3. \( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L \land Result \neq L \land Result \leq c\} \)

Result:=Result+1

\( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L\} \) (CONS)((\(v \neq a(Result)\)) implies \((Result \in Ps(a,c,v))\), This together with the first condition implies implies Result \neq L)

4. \( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L \land (v \neq a(Result)) \land Result \leq c\} \)

Result:=Result+1

\( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L\} \)

(CONS)((\(v \neq a(Result)\)) implies \((Result \in Ps(a,c,v))\), This together with the first condition implies implies Result \neq L)

5. \( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L\} \)

while (Result <= c) and not (\(v = a(Result)\)) do Result:=Result+1 end

\( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L \land (v = a(Result) \lor Result > c)\} \)

6. \( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L\} \)

while (Result <= c) and not (\(v = a(Result)\)) do Result:=Result+1 end

\( \{Ps(a,c,v) = \{L\} \land 1 \leq Result \leq L \land (v = a(Result)) \land L \leq c\} \) (CONS)
7. \{ P_s(a, c, v) = \{L\} \land 1 \leq Result \leq L \} \\
while (Result \leq c) and not (v = a(Result)) do \\
Result := Result + 1 end;

\{ P_s(a, c, v) = \{L\} \land Result \leq c \land (v = a(Result)) \} (CONS) \\

8. \{ P_s(a, c, v) = \{L\} \land 1 \leq Result \leq L \} \\
while (Result \leq c) and not (v = a(Result)) do Result := Result + 1 end;

\{ L = Result \land Result \leq c \} (CONS) \\

9. \{ P_s(a, c, v) = \{L\} \land 1 \leq 1 \leq L \} \\
Result := 1 \\

\{ P_s(a, c, v) = \{L\} \land 1 \leq Result \leq L \} (ASSN) \\

10. \{ P_s(a, c, v) = \{L\} \land 1 \leq 1 \leq L \} \\
Result := 1; while (Result \leq c) and not (v = a(Result)) do Result := Result + 1 end;

\{ L = Result \land Result \leq c \} (COMP) \\

11. \{ P_s(a, c, v) = \{L\} \} \\
Result := 1; while (Result \leq c) and not (v = a(Result)) do Result := Result + 1 end;

\{ L = Result \land Result \leq c \} (CONS) \\

Proof for Specification 2:

1. \{ P(a, c, v) = \{\} \land Result + 1 \geq 1 \} \\
Result := Result + 1 \\
\{ P(a, c, v) = \{\} \land Result \geq 1 \}
2. \( \{ P(a, c, v) = \{ \} \land \text{Result} \geq 1 \} \)

\[ \text{Result} := \text{Result} + 1 \]

\( \{ P(a, c, v) = \{ \} \land \text{Result} \geq 1 \} \) (CONS)

3. \( \{ P_s(a, c, v) = \{ \} \land \text{Result} \geq 1 \land (v \neq a(\text{Result})) \land \text{Result} \leq c \} \)

\[ \text{Result} := \text{Result} + 1 \]

\( \{ P(a, c, v) = \{ \} \land \text{Result} \geq 1 \} \) (CONS)

4. \( \{ P(a, c, v) = \{ \} \land \text{Result} \geq 1 \} \)

\[ \text{while} \ (\text{Result} \leq c) \text{ and not } (v = a(\text{Result})) \text{ do } \text{Result} := \text{Result} + 1 \text{ end;} \]

\( \{ P(a, c, v) = \{ \} \land \text{Result} \geq 1 \land (\text{Result} > c \lor v = a(\text{Result})) \} \) (WHILE)

5. \( \{ P(a, c, v) = \{ \} \} \) Result:=1 \( \{ P(a, c, v) = \{ \} \land \text{Result} \geq 1 \} \) (ASSN)

6. \( \{ P(a, c, v) = \{ \} \} \)

\[ \text{Result} := 1; \text{while} \ (\text{Result} \leq c) \text{ and not } (v = a(\text{Result})) \text{ do } \text{Result} := \text{Result} + 1 \text{ end;} \]

\( \{ P(a, c, v) = \{ \} \land \text{Result} \geq 1 \land (\text{Result} > c \lor v = a(\text{Result})) \} \)

(COMP, 4, 5)

7. \( \{ P(a, c, v) = \{ \} \} \)

\[ \text{Result} := 1; \text{while} \ (\text{Result} \leq c) \text{ and not } (v = a(\text{Result})) \text{ do } \text{Result} := \text{Result} + 1 \text{ end;} \]

\( \{ \text{Result} > c \} \) (CONS, 6)(Details are omitted)

Here we have an example of some code which should terminate in a state where \( b1 \) is true and \( b2 \) is false:
$k : \text{SET}[\text{INTEGER}]$;
$b1, b2 : \text{BOOLE};$
$k.\text{create}();$
$k.\text{include}(78);$
$b1 := k.\text{contains}(78);$
$b2 := k.\text{contains}(65);$ 

To make the proof short we use $Z$ to represent the condition

$k.\text{type} = \text{SET}[\text{INTEGER}]$

$\{\text{type}(k) = \text{SET}[\text{INTEGER}]\}$

$k.\text{create}();$

$\{k.\text{count} = 0 \land k.\text{type} = \text{SET}[\text{INTEGER}]\}$

$\{I(k.a, k.c) \land k.\text{type} = \text{SET}[\text{INTEGER}]\}$

$\{I(k.a, k.c) \land Ps(k.a, k.c, 65) = \emptyset \land Z\}$

$k.\text{include}(78);$

$\{\exists n (Ps(k.a, k.c, 65) = \{n\}) \land Ps(k.a, k.c, 65) = \emptyset \land Z\}$

$b1 := k.\text{contains}(78);$

$\{b1 \land Ps(k.a, k.c, 65) = \emptyset \land Z\}$

$b2 := k.\text{contains}(65);$

$\{b1 \land \neg b2\}$

Note that we did not explicitly use any of the procedure call axioms of the original proof system. It is easier to prove the program solely from the specification.
Chapter 6  VERIFICATION OF PROGRAMS
WITH EXCEPTION HANDLING

Section 1 Introduction

Unexpected events can occur during the execution of a program. Running out of memory and zero-divisor are a couple examples. Exceptions will normally cause a program to terminate with an appropriate error message. Exception handling serves the purpose of allowing a program to continue execution in the event of an exception. Handling exceptions, however, can result in complex looking source code. Several languages, including C++ and Eiffel, offer exception handlers as a solution to this problem. Exception handlers can be more error prone than other constructs because exceptions can occur at almost any point in the flow of control and an exception causes a jump in the flow of control. Exception handlers are normally used to take care of unexpected events, however, if an exception occurs as the result of a programming error then unexpected results are likely to occur and not be handled properly. This is why it is important to ensure the validity of programs that use exception handlers.

An article by Eugene Seals (See [77]) advocates the use of exception reporting in computer performance analysis and shows an example using actual data from production facilities. This is a predecessor of exception handling. The exception handling of Eiffel is described in [63] and [69]. Chapter 3 of [53] has extensive
material on the exception handling of C++ along with many examples to illustrate the usage of the exception handling constructs. Chapter 15 of [76] and Chapter 15 of [27] also contain some material on the exception handling of C++. Program verification, without the presumption that all expressions are valid, can be done using something called free arithmetic (See chapter 5 of [37]). It does not appear, however, that exception handling routines for such situations are covered.

The subject of this chapter is the verification of programs that use exception handling constructs. A mini-language and a proof system is developed for proving programs in this mini-language. Exceptions resulting from evaluation of invalid expressions, violation of assertions and running out of memory are considered. Exceptions thrown on purpose are also covered. The deductive rules and axioms in EPLW are designed to be mechanical and non-trivial to use. Some object-oriented extensions to EPLW are also given.

Section 2 Preliminaries

An exception handler is a routine that accepts control in the event of an exception. Usually an exception handler will either restore the integrity of the program and continue or will terminate the program without any catastrophic events occurring.

In Eiffel an exception handler can be attached to a routine. The rescue clause of a routine, if there is one, will be called whenever an exception occurs during
the routine's execution. Two causes of Eiffel exceptions are assertion violations and signals triggered by the operating system. The rescue clause of a routine can execute a retry instruction which restarts the execution of the routine. If no retry instruction is executed in the rescue clause, the routine will fail. An exception is triggered in the calling routine whenever a routine fails.

If an exception occurs during the execution of a routine with no rescue clause, a default rescue procedure is called. This procedure will normally do nothing and thus cause the routine to fail. A class can override this by providing a specific rescue procedure. Every routine in Eiffel has an explicit, or implicit, rescue clause.

The exception handler in this Eiffel routine sets tried to true and retries the routine. An attempt to divide by zero will cause the exception, if there is one.

```eiffel
divide(a:REAL,b:REAL) is
  local
    tried:BOOLEAN;
  do
    if tried then
      success := false;
    else
      answer := a/b;
      success := true;
    end;
    rescue
      tried := true;
      retry;
  end;
```

The EXCEPTIONS kernel library class for Eiffel provides tools for fine tuning the exception mechanism. This class provides an integer code for every possible type of exception and a variable to hold the exception code. It also provides a routine for raising exceptions on purpose.
The language of Eiffel includes several types of assertions which can be monitored during the execution of a program. The assertions in Eiffel include preconditions, postconditions, loop invariants, loop variants, check instructions and class invariants. Assertions can be valuable in debugging programs however there are currently no techniques to verify that an Eiffel assertion is always satisfied.

In C++ the exception handling mechanism has a different flavor to it. An exception handler is not attached to a routine but to a try block. A try block constitutes a section of a program that is subject to exception checking. A handler is invoked by a throw command executed within a try block or within a function called by the try block. The argument to the throw command is passed down to the exception handler. A handler is a catch function. A set of one or more catch functions appears immediately after every try block. The class of a thrown expression determines which exception handler to invoke. The easiest way to define a signal to throw is to define a class with no members in it, then use the name of the class as the argument to the throw command. Whenever a throw statement is executed, if the appropriate catch statement exists, it will be invoked, otherwise a terminate routine is called. This terminate routine can be changed by the programmer. This C++ code does the same thing as the Eiffel routine except that the zerodivide is explicitly checked.

class zerodivide { }
void divide(float a, float b, float c, int& success)
try
  if (b=0) throw(zerodivide());
  c = a/b;
  success = 1;
}
catch(zerodivide)
  success = 0;

**Definition 6.2.1** For all boolean expressions $B$ and all expressions $c$ and $d$:

$$(B?c : d) = \begin{cases} c & \text{if } B \text{ is true} \\ d & \text{if } B \text{ is false} \end{cases}$$

Below are several properties of the conditional function. Proofs of these properties are omitted. To prove any one of them you simply need to consider the two cases of $B = \text{true}$ and $B = \text{false}$.

1. $$(B?(B!c : d) : e) = (B!c : e) \text{ and } (B!e : (B!c : d)) = (B!e : d)$$
2. $$(B?f(B, \bar{x}) : g(B, \bar{y})) = (B?f(\text{true}, \bar{x}) : g(\text{false}, \bar{y}))$$
3. $$(B!e : e) = e$$
4. If $B$, $C$ and $D$ are boolean expressions then
   a. $$(B?C : D) = ((B \land C) \lor (\neg B \land D)) \text{ (Here you can avoid writing the extra copy of the boolean expression B)}$$
   b. $$(B?\text{false} : C) = \neg B \land C \text{ and } (B?C : \text{false}) = B \land C$$
   c. $$(B \land C) \Rightarrow (B?C : D) \text{ and } (\neg B \land D) \Rightarrow (B?C : D)$$

The expression $P_{e^x}$, a very commonly used syntax, can be obtained from $P$ by substituting $e$ for every occurrence of $x$. The expression $P_{e_1,...,e_n}$ can be obtained from $P$ by simultaneously substituting $e_i$ for every occurrence of $x_i$ for $i = 1...n$. 80
Section 3 The mini-language EPLW

This section begins with the introduction of a system variable called \textit{excode}. It is assigned \textit{nil} in any state where there is no pending exception. In any state with a pending exception the appropriate exception code is assigned to \textit{excode}.

Below is the syntax for the mini-language \textit{EPLW (Exception-PLW)} with square brackets indicating optional items and $K^*$ representing 0 or more $K$'s where $K$ is any nonterminal.

\begin{align*}
S & := S; S \mid x := c \mid \textit{skip} \mid \textit{if } B \textit{ then } S \textit{ else } S \textit{ end} \mid \textit{while } B \textit{ do } S \textit{ end} \\
& \quad \mid \textit{try } S \ H \mid \textit{throw}(c) \mid \textit{canthrow}(c) \mid \textit{assert}(c) \ B \textit{ end} \\
& \quad \mid x, \textit{forget}; \mid y := x;
\end{align*}

\begin{align*}
E & := c, E \mid c \\
H & := \textit{catch}(c) S \textit{ end } H \mid \textit{catch } S \textit{ end } \mid \textit{catchproc}(c) S \textit{ end}
\end{align*}

The try block is defined in terms of the catch construct and the catch construct is included in the configuration definition. A catch construct is not an $S$ construct so that a catch construct can only catch exceptions thrown from within the try block. There are some rules associated with the mini-language. 1) There is a system variable called \textit{excode} which is to hold the exception code to pass down to an exception handler. 2) An \textit{EPLW} program is syntactically not to contain the system variable \textit{excode}. 3) A state $\sigma$ is said to have a pending exception if and only if $\sigma[\textit{excode}] \neq \textit{nil}$. 4) Any construct that is not an exception handler is
to terminate in its initial state whenever its initial state has a pending exception. This simulates the jumping to the next available exception handler in the event of an exception.

**Definition 6.3.1** The transition relation on configurations for the mini-language **EPLW** consists of the following:

1. For states $\sigma$ where $\sigma[\text{excode}] = \textit{nil}$ we have the following:
   
   a. $\langle x := e, \sigma \rangle \rightarrow \langle \lambda, \sigma[e/x] \rangle$
   
   b. $\langle \textit{skip}, \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle$
   
   c. $\langle \text{while } B \text{ do } S \text{ end }, \sigma \rangle \rightarrow \langle S; \text{while } B \text{ do } S \text{ end }, \sigma \rangle$ if $\sigma[B] = \textit{true}$

   $\langle \text{while } B \text{ do } S \text{ end }, \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle$ if $\sigma[B] = \textit{false}$
   
   d. $\langle \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end }, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle$ if $\sigma[B] = \textit{true}$

   $\langle \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end }, \sigma \rangle \rightarrow \langle S_2, \sigma \rangle$ if $\sigma[B] = \textit{false}$
   
   e. $\langle \text{proc } N(V_1) K, \sigma \rangle \rightarrow \langle \lambda, \sigma[\text{proc } N(V_1) K|N] \rangle$
   
   f. $\langle N(A_2), \sigma \rangle \rightarrow \langle S^{A_1}_{A_2}, \sigma \rangle$ if $\sigma[N] = \text{proc } N(A_1) \text{ do } S \text{ end}$

   $\langle N(A_2), \sigma \rangle \rightarrow \langle S^{A_1,v_1,...,v_n}_{A_2,ls,x_1,...,ls,v_n}, \sigma \rangle$

   if $\sigma[N] = \text{proc } N(A_1) \text{ local } v_1...v_n \text{ do } S \text{ end}$

   where $ls$ is a newly created temporary variable.

   g. $\langle \text{try } S H, \sigma \rangle \rightarrow \langle S H, \sigma \rangle$

   h. $\langle \text{throw}(e), \sigma \rangle \rightarrow \langle \lambda, \sigma[e|\text{excode}] \rangle$
i. \[(\text{canthrow}(c).\sigma) \rightarrow (\lambda.\sigma[c|\text{excode}])\]
\[(\text{canthrow}(c).\sigma) \rightarrow (\lambda.\sigma)\]

j. \[(\text{assert}(c.B \text{ end}.\sigma) \rightarrow (\lambda.\sigma) \text{ if } \sigma[B] = \text{true}\]
\[(\text{assert}(c.B \text{ end}.\sigma) \rightarrow (\lambda.\sigma[c|\text{excode}]) \text{ if } \sigma[B] = \text{false}\]

k. \[(\text{catch}(c.S \text{ end}.\sigma) \rightarrow (\lambda.\sigma)\]
\[(\text{catch} \text{ end}.\sigma) \rightarrow (\lambda.\sigma)\]
\[(\text{catch} \text{ hproc}(c) \text{ end}.\sigma) \rightarrow (\lambda.\sigma)\]

2. For states \(\sigma\) where \(\sigma[\text{excode}] \neq \text{nil}\) we have

a. \((S.\sigma) \rightarrow (\lambda.\sigma)\) for every \(S\) construct.

b. \((\text{catch}(c) S \text{ end}.\sigma) \rightarrow (S.\sigma[\text{nil}|\text{excode}])\) if \(\sigma[\text{excode}] = c\)
\[(\text{catch}(c) S \text{ end}.\sigma) \rightarrow (\lambda.\sigma) \text{ otherwise.}\]

c. \((\text{catch} S \text{ end}.\sigma) \rightarrow (S.\sigma[\text{nil}|\text{excode}])\)

d. \((\text{catch} \text{ hproc}(c) S \text{ end}.\sigma) \rightarrow (S.\sigma[\text{excode}|c][\text{nil}|\text{excode}])\)

3. If \((S.\sigma) \rightarrow (S'.\sigma')\) then for all \(T (S;T.\sigma) \rightarrow (S';T.\sigma')\)

\[\square\]

The semantics are illustrated by the following computation starting in a state
without a pending exception:

\[
\begin{align*}
\langle \text{throw}(8); x := x + y; y := y + 1; z := x + y; \text{catch}(8) x := c; \text{end}, \sigma \rangle & \rightarrow \\
\langle x := x + y; y := y + 1; z := x + y; \text{catch}(8) x := c; \text{end}, \sigma[8]\text{decode} \rangle & \rightarrow \\
\langle y := y + 1; z := x + y; \text{catch}(8) x := c; \text{end}, \sigma[8]\text{decode} \rangle & \rightarrow \\
\langle z := x + y; \text{catch}(8) x := c; \text{end}, \sigma[8]\text{decode} \rangle & \rightarrow \\
\langle \text{catch}(8) x := c; \text{end}, \sigma[8]\text{decode} \rangle & \rightarrow \\
\langle x := c, \sigma[8]\text{decode}[\text{nil}\text{decode}] \rangle & \rightarrow 
\end{align*}
\]

Since \(\sigma[8]\text{decode}[\text{decode}] \neq \text{nil}\), meaning that there is a pending exception, each of the assignments perform no operation and therefore the catch statement is the next statement to affect the state of the computer.

**Section 4 Proof System for EPLW**

**Definition 6.4.1** Let \(Ex\) stand for \((\text{decode} \neq \text{nil})\) meaning that there is a pending exception.

Whenever we need to state an ordinary condition in the form \((Ex?P : Q)\) we can use the rule \(P = (Ex?P : P)\).

(SKIP)

\[
\{P\} \text{skip}\{P\}
\]

(ASSN)

\[
\{Ex?P : Q\}_\sigma x := e\{Ex?P : Q\}
\]

84
(WHILE)

\[ \{ \text{Ex}? \text{false} : (P \land B) \} S \{ P \} \]
\[ \{ P \} \text{while B do S end} \{ \text{Ex}? P : (P \land \neg B) \} \]

(IF)

\[ \{ \text{Ex}? X_1 : X_2 \} S_1 \{ Q \} \land \{ \text{Ex}? Y_1 : Y_2 \} S_2 \{ Q \} \]
\[ \{ \text{Ex}? Q : (B?X_2 : Y_2) \} \text{if B then S_1 else S_2 end} \{ Q \} \]

(PROC) For procedure proc \( N(A_1) \) do S end we have

\[ \{ \text{Ex}? X : P \} S_{A_2}^{A_1} \{ Q \} \]
\[ \{ \text{Ex}? Q : P \} N(A_2) \{ Q \} \]

(PROCL) For proc \( N(A_1) \) local \( v_1...v_n \) do S end and \( ls \) not occurring in \( Q \) we have

\[ \{ \text{Ex}? X : P \} S_{A_2,ls,v_1...v_n}^{A_1,v_1...v_n} \{ Q \} \]
\[ \{ \text{Ex}? Q : P \} N(A_2) \{ Q \} \]

(TRY)

\[ \{ \text{Ex}? X : Y \} S H \{ Q \} \]
\[ \{ \text{Ex}? Q : Y \} \text{try S H} \{ Q \} \]

(CATCH)

\[ \{ \text{Ex}? X : Y \} S \{ Q \} \]
\[ \{ \text{Ex}? ((\text{excode} = c)?Y^{\text{excode}} : Q) : Q \} \text{catch(c) S end} \{ Q \} \]

(CATCH-DEF)

\[ \{ \text{Ex}? X : Y \} S \{ Q \} \]
\[ \{ \text{Ex}? Y^{\text{excode} : Q} \} \text{catch S end} \{ Q \} \]

(CATCH-PROC)

\[ \{ \text{Ex}? X : Y \} S \{ Q \} \]
\[ \{ \text{Ex}? (Y^{\text{excode} : Q})_c \} \text{catchproc(c) S end} \{ Q \} \]
(THROW)
\[
\left\{ \text{Ex!} P : P_{e}^{\text{encode}} \right\}^{\text{throw}(e)} \{ \text{Ex?} P : Q \}
\]

(CANTHROW)
\[
\left\{ \text{Ex?} P : \left( Q \land P_{e}^{\text{encode}} \right) \right\}^{\text{canthrow}(e)} \{ \text{Ex?} P : Q \}
\]

(ASSERT)
\[
\left\{ \text{Ex?} P : \left( B? Q : \left( P_{e}^{\text{encode}} \right) \right) \right\}^{\text{assert}(c) \ B \ end} \{ \text{Ex?} P : Q \}
\]

(CONS)
\[
P_{1} \Rightarrow P_{2} \land \left\{ P_{2} \right\} S \{ Q_{2} \} \land Q_{2} \Rightarrow Q_{1} \\
\{ P \} S \{ Q_{1} \}
\]

(COMP)
\[
\{ P \} S_{1} \{ Q \} \land \{ Q \} S_{2} \{ R \} \\
\{ P \} S_{1} : S_{2} \{ R \}
\]

(AUX1) For all $S$ programs $S$
\[
\left\{ \neg \text{Ex} \land P \right\} S \{ Q \} \\
\{ \text{Ex?} Q : P \} S \{ Q \}
\]

(AUX2) If $A = (\text{Ex?} P : Q)$, $c \neq \text{nil}$, $y \neq \text{encode}$, and $x \neq \text{encode}$ then
\[
\text{Ex?} A : f \left( A, A_{e}^{\text{encode}}, A_{y}^{\text{x}} \right) \equiv \text{Ex?} P : f \left( Q, P_{e}^{\text{encode}}, Q_{x}^{y} \right)
\]

(JUMP) For every $S$ program and predicate $P$ the following specifications hold
\[
\left\{ \text{Ex} \land P \right\} S \{ \text{Ex} \land P \}
\]

86
\{ E : x \land P \} S \{ P \}

A reminder that a fragment of code with a catch keyword, which is not part of a try construct, is not a member of S.

(OR)

\frac{\{ P_1 \} S \{ Q_1 \} \land \{ P_2 \} S \{ Q_2 \}}{\{ P_1 \lor P_2 \} S \{ Q_1 \lor Q_2 \}}

(AND)

\frac{\{ P_1 \} S \{ Q_1 \} \land \{ P_2 \} S \{ Q_2 \}}{\{ P_1 \land P_2 \} S \{ Q_1 \land Q_2 \}}

The rule for iteration is much different because it is now possible for B to be true upon termination of the loop. This is because an exception could occur in S and we must terminate at that time even if B is still true.

The following algorithm is written in the mini-language EPLW. It is a single try construct that contains code to perform a division. If the divisor is 0 then an exception is thrown and an appropriate exception handler is called.

```
try
    if b=0 then
        throw(8)
    else
        c:=a/b
    end;
catch proc(p)
    X:=p;
end;
```

We claim that the above algorithm is correct according to precondition \{ \neg E : x \} and postcondition \{ c = a/b \lor x = 8 \}. Note that no exception is pending in the precondition otherwise no operations would be performed.
1. \{Ex?(c = a/b ∨ x = 8) : (c = a/b ∨ p = 8)\}
   
   \begin{align*}
   &x := p \\
   &\{c = a/b ∨ x = 8\} \quad \text{(ASSN)}
   \end{align*}
2. \{Ex?(c = a/b ∨ excode = 8) : (c = a/b ∨ x = 8)\}
   
   catchproc(p) x := p end
   
   \{c = a/b ∨ x = 8\} \quad \text{(CATCH-PROC)}
3. \{Ex?(c = a/b ∨ excode = 8) : (c = a/b ∨ 8 = 8)\}
   
   throw(8)
   
   \{Ex?(c = a/b ∨ excode = 8) : (c = a/b ∨ x = 8)\} \quad \text{(THROW)}
   
   The precondition can be simplified to \{Ex?(c = a/b ∨ excode = 8) : true\}
4. \{Ex?(c = a/b ∨ excode = 8) : (a/b = a/b ∨ x = 8)\}
   
   c := a/b
   
   \{Ex?(c = a/b ∨ excode = 8) : (c = a/b ∨ x = 8)\} \quad \text{(ASSN)}
   
   The precondition can be simplified to \{Ex?(c = a/b ∨ excode = 8) : true\}
5. \{Ex?Q : ((b = 0)?true : true)\}
   
   \text{if } b = 0 \text{ then throw(8) else } c := a/b \text{ end}
   
   \{Ex?(c = a/b ∨ excode = 8) : (c = a/b ∨ x = 8)\}
   
   where \( Q = (Ex?(c = a/b ∨ excode = 8) : (c = a/b ∨ x = 8)) \) \quad \text{(IF,3,4)}
   
   The precondition can be simplified to \{Ex?(c = a/b ∨ excode = 8) : true\}
6. \{Ex?(c = a/b ∨ excode = 8) : true\}
   
   \text{if } b = 0 \text{ then throw(8) else } c := a/b \text{ end; catch(p) } x := p \text{ end}
\{ c = a/b \lor x = 8 \} \quad \text{(COMP)}

7. \{ Ex?(c = a/b \lor x = 8) : true \}

\text{try if } b = 0 \text{ then throw(8) else } c := a/b \text{ end catchproc(p) } x := p \text{ end}

\{ c = a/b \lor x = 8 \} \quad \text{(TRY)}

8. \{ \neg Ex \}

\text{try if } b = 0 \text{ then throw(8) else } c := a/b \text{ end catch(p) } x := p \text{ end}

\{ c = a/b \lor x = 8 \} \quad \text{(CONS)}

**Section 5 Soundness of Proof System**

**Definition 6.5.1** Program segment \( S \) is partially correct according to precondition \( P \) and postcondition \( Q \), denoted \( \{ P \}S\{ Q \} \), if and only if
\[
(\sigma \models P \land \langle S, \sigma \rangle \rightarrow (\lambda, \sigma')) \Rightarrow (\sigma' \models Q).
\]

**Theorem 6.5.2** The axioms associated with \( EPLW \) are valid specifications and the deductive rules associated with \( EPLW \) preserve the validity of specifications.

\[\square\]

**Proof:**

(AUX1)

1. Suppose \( \{ \neg Ex \land P \}S\{ Q \} \) for an \( S \) program.
2. Suppose \( \sigma \models Ex \land Q \)
3. \( \langle S, \sigma \rangle \rightarrow (\lambda, \sigma) \) since we are dealing with an \( S \) program.
4. \( \{ Ex \land Q \}S\{ Q \} \) since \( \sigma \models Q \)
5. \{E:x\land P\}S\{Q\} (1,4,OR)

(AUX2)

1. \(E:x?A : f(A, A_c^{encode}, A_r^y)\)
3. \(E:x!P : f(Q, ((E:x)_c^{encode}, P_r^{encode} : Q_r^{encode}).((E:x)_r^y, P_r^y : Q_r^y))\)
4. \(E:x!P : f(Q, ((\sigma = \text{nil})?P_c^{encode} : Q_c^{encode}).(E:x!P_r^y : Q_r^y))\)
5. \(E:x!P : f(Q, P_c^{encode}, Q_r^y)\)

(JUMP)

1. Suppose \(\sigma \models E:x \land P\) for some predicate \(P\).
2. \(\langle S, \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle\) for every \(S\) construct (from transition relation).
3. \(\langle S_n; S_{n-1}; \ldots; S_1, \sigma \rangle \rightarrow \ldots \rightarrow \langle S_3; S_2; S_1, \sigma \rangle \rightarrow \langle S_2; S_1, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle\)
   where \(S_1 \ldots S_n\) are \(S\) constructs.
4. \(\{E:x \land P\}S\{E:x \land P\}\) for every \(S\) program (definition of partial correctness)
5. \(\{E:x \land P\}S\{P\}\) for every \(S\) program (CONS)

(SKIP) Let us prove this by cases. 1) If there is an exception then, because the
statement is not a catch statement, the state of the program is not changed. 2)
When there is no error pending, the state of the program is still not changed.
Therefore the operation skip will never change the state of a program. The axiom
has been proven.
(ASSN)

1. Suppose: \( \sigma \models \neg E x \land A^x_r \)

2. \( \langle x := e, \sigma \rangle \rightarrow \langle \lambda. c[x], e \rangle \) from the configuration.

3. \( \{ \neg E x \land A^x_r \} x := e \{ K \} \) where \( \sigma[c[x]] \models K \) and thus \( \sigma \models K^x_r \)

4. \( (\neg E x \land A^x_r)^x_r = (\neg E x)^x_r \land A^x_r = \neg E x \land A^x_r \)

5. \( \{ \neg E x \land A^x_r \} x := e \{ \neg E x \land A \} \)

6. \( \{ \neg E x \land A^x_r \} x := e \{ A \} \)

7. \( \{ E x ? A : A^x_r \} x := e \{ A \} \) (AUX1,x:=e is an S program)

8. \( \{ E x ? P : Q^x_r \} x := e \{ E x ? P : Q \} \)

(WHILE) Suppose \( \{ E x ? f a l s e : ( P \land B ) \} S \{ P \} \)

1. Let us consider the following computation of while B do S end:

\( C_0 \overset{*}{\rightarrow} C_1 \overset{*}{\rightarrow} \ldots \overset{*}{\rightarrow} C_n \) where for all \( 0 \leq i \leq n \) \( C_i \rightarrow \ldots \rightarrow C_{i+1} \) represents a computation of \( S \), \( C_i = \langle \text{while } B \text{ do } S \text{ end}, \sigma_i \rangle \), \( C_n = \langle \lambda. \sigma_n \rangle \) and \( \sigma_0 \models P \).

2. Here we prove \( \sigma_i \models P \Rightarrow \sigma_{i+1} \models P \) for all \( 0 \leq i \leq n \).

a. \( \sigma_i \models P \)

b. \( \sigma_i \models P \land B \land \neg E x \) since we are still in the while loop.

c. \( \sigma_i \models E x ? f a l s e : ( P \land B ) \) (equivalence of expressions)

d. \( \sigma_{i+1} \models P \) (1, antecedent, definition of partial correctness)
3. $\sigma_i \models P$ for all $0 \leq i \leq n$. (2, math induction)

4. $\sigma_n \models P$

5. $\sigma_n \models P \land (\exists x \lor \neg B)$

6. $\sigma_n \models \exists x \exists y : (P \land \neg B)$

(IF) Suppose $\{\exists x X_1 : X_2\} S_1 \{Q\} \land \{\exists x Y_1 : Y_2\} S_2 \{Q\}$

1. Suppose $\sigma \models \neg \exists x \land B \land X_2$
   a. $\sigma \models \exists x X_1 : X_2$
   b. $(\text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end}) \rightarrow (S_1, \sigma) \xrightarrow{\cdot} (\lambda, \sigma_2)$ for $\sigma_2 \models Q$
   c. $\{\neg \exists x \land B \land X_2\} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end} \{Q\}$

2. Similarly we can prove $\{\neg \exists x \land \neg B \land Y_2\} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end} \{Q\}$

3. $\{(\neg \exists x \land B \land X_2) \lor (\neg \exists x \land \neg B \land Y_2)\} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end} \{Q\}$

(1, 2, OR)

4. $\{\neg \exists x \land (B?X_2 : Y_2)\} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end} \{Q\}$ (simplification)

5. $\{\exists x?Q : (B?X_2 : Y_2)\} \text{if } B \text{ then } S_1 \text{ else } S_2 \text{ end} \{Q\}$ (AUX1)

(TRY) Assume $\{\exists x?X : Y\} S \ H \{Q\}$

1. Suppose $\sigma \models \neg \exists x \land Y$

2. $(\text{try } S \ H, \sigma) \rightarrow (S \ H, \sigma)$ from configuration.

3. $\sigma \models \exists x?X : Y$

4. $(S \ H, \sigma) \xrightarrow{\cdot} (\lambda, \sigma_1)$ for some $\sigma_1 \models Q$

5. $\{\neg \exists x \land Y\} \text{try } S \ H \{Q\}$
6. \{Ex?Q : Y\}try S H\{Q\} (AUX1)

(CATCH-PROC) Suppose \{Ex?X : Y\}S\{Q\}

1. Suppose \(\sigma \models Ex \land (Y_{\text{encode}})^c_{e,\text{encode}}\)
   a. \(\sigma \models (Y_{\text{encode}})^c_{e,\text{encode}}\)
   b. \(\sigma[e,\text{encode}][\text{nil}[\text{encode}]] \models Y\)
   c. \(\sigma[e,\text{encode}][\text{nil}[\text{encode}]] \models \neg Ex \land Y\) since \text{encode} is \text{nil} in this state.
   d. \(\sigma[e,\text{encode}][\text{nil}[\text{encode}]] \models Ex?X : Y\)
   e. \((\text{catchproc}(c)\ S\ \text{end,\ }\sigma) \rightarrow (S, \sigma[e,\text{encode}][\text{nil}[\text{encode}]]) \rightarrow (\lambda, \sigma')\) for some \(\sigma' \models Q\)
   f. \(\{Ex \land (Y_{\text{encode}})^c_{e,\text{encode}}\}\text{catchproc}(c)\ S\ \text{end}\{Q\}\)

2. Suppose \(\sigma \models \neg Ex \land Q\)
   a. \((\text{catchproc}(c)\ S\ \text{end,\ }\sigma) \rightarrow (\lambda, \sigma)\)
   b. \(\{\neg Ex \land Q\}\text{catchproc}(c)\ S\ \text{end}\{Q\}\) since \(\sigma \models Q\)

3. \(\{Ex? (Y_{\text{encode}})^c_{e,\text{encode}} : Q\}\text{catchproc}(c)\ S\ \text{end}\{Q\}\) (1,2,OR)

(Proofs of CATCH and CATCH-DEF will be similar to the proof for CATCH-PROC and therefore they are omitted)

(THROW)

1. Suppose \(\sigma \models \neg Ex \land P_{e,\text{encode}}\)
   a. \(\sigma \models P_{e,\text{encode}}\)

93
b. \( \sigma[c|{\text{execute}}] \models P \)

c. \( \{ \neg \text{Ex} \land P_c^{\text{execute}} \} \text{throw}(c) \{ P \} \)

2. \( \{ \text{Ex}?: P : P_c^{\text{execute}} \} \text{throw}(c) \{ P \} \) (AUX1)

3. \( \{ \text{Ex}?: P : P_c^{\text{execute}} \} \text{throw}(c) \{ \text{Ex}?: P : Q \} \) (AUX2)

(CANTHROW)

1. Suppose \( \sigma \models \neg \text{Ex} \land A \land A_c^{\text{execute}} \)

Since we have nondeterminism with two possible computations we need to prove that the final outcome is valid with both computations.

a. Case 1: \( \langle \text{canthrow}(c), \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle \)

\[ \begin{align*}
\sigma & = \sigma' \\
\sigma' & \models \neg \text{Ex} \land A \land A_c^{\text{execute}} \\
\sigma' & \models A
\end{align*} \]

b. Case 2: \( \langle \text{canthrow}(c), \sigma \rangle \rightarrow \langle \lambda, \sigma[c|{\text{execute}}] \rangle \)

\[ \begin{align*}
\sigma' & = \sigma[c|{\text{execute}}] \\
\sigma & \models A_c^{\text{execute}} \text{ therefore } \sigma' \models A
\end{align*} \]

Subconclusion: \( \{ \neg \text{Ex} \land A \land A_c^{\text{execute}} \} \text{canthrow}(c) \{ A \} \)

2. \( \{ \text{Ex}?: A : (A \land A_c^{\text{execute}}) \} \text{canthrow}(c) \{ A \} \) (AUX1)

3. \( \{ \text{Ex}?: P : (Q \land P_c^{\text{execute}}) \} \text{canthrow}(c) \{ \text{Ex}?: P : Q \} \) (AUX2)

94
(ASSERT)

1. Suppose: \( \sigma \models \neg Ex \land B \land A \)
   a. \( \langle assert(c) B \ end, \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle \) from configuration.
   b. \( \{\neg Ex \land B \land A\} assert(c) B \ end \{\neg Ex \land B \land A\} \)
   c. \( \{\neg Ex \land B \land A\} assert(c) B \ end \{A\} \) (CONS)

2. Suppose \( \sigma \models \neg Ex \land \neg B \land A_{cexcode}^c \)
   a. \( \langle assert(c) B \ end, \sigma \rangle \rightarrow \langle \lambda, \sigma[c\lceil excode \rceil] \rangle \) from configuration.
   b. \( \sigma \models A_{cexcode}^c \) therefore \( \sigma[c\lceil excode \rceil] \models A \)
   c. \( \{\neg Ex \land \neg B \land A_{cexcode}^c\} assert(c) B \ end \{A\} \)

3. \( \{\neg Ex \land (B?A : A_{cexcode}^c)\} assert(c) B \ end \{A\} \) (1,2.OR)

4. \( \{Ex?A : (B?A : A_{cexcode}^c)\} assert(c) B \ end \{A\} \) (AUX1)

5. \( \{Ex?P : (B?Q : P_{cexcode}^c)\} assert(c) B \ end \{Ex?P : Q\} \) (AUX2)

(PROC) The antecedent is \( \{Ex?X : P\} S_{A_{c}}^{A_{1}} \{Q\} \). From this we prove the consequent.

1. Assume \( \sigma \models Ex \land Q \)
   a. \( \langle N(A_{2}), \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle \) (definition of transition relation)
   b. \( \{Ex \land Q\} N(A_{2}) \{Q\} \)

2. Assume \( \sigma \models \neg Ex \land P \)
   a. \( \langle N(A_{2}), \sigma \rangle \rightarrow \langle S_{A_{c}}^{A_{1}}, \sigma \rangle \) (Definition of transition relation)
b. $\sigma \vdash Fx.X : P$ for some $X$ (implied by assumption)

(c. $\left< S_{A_1}^{A_2}, \sigma \right> \vdash \left< \lambda, \sigma' \right>$ for some $\sigma' \models Q$ (from antecedent)

(d. $\{ \neg Fx \land P \} N(A_2) \{ Q \}$

3. $\{ Fx : Q \} N(A_2) \{ Q \}$ (1.2.OR)

(PROCL) Proof is the same as the proof of (PROC) except $S_{A_1}^{A_2}$ is replaced with $S_{A_2 \wedge \ldots}^{A_2 \wedge \ldots}$

Section 6 Exceptions in Expressions

Situations where the evaluation of an expression (integer expression, boolean expression etc) causes an exception are considered in this section. For all expressions I define a function $exc(e)$. This function returns true whenever an obvious exception will occur in the evaluation of the expression $e$. Here is a possible definition for the function for integer expressions:

$exc(e_1 + e_2) = (|e_1 + e_2| > maxint) \lor exc(e_1) \lor exc(e_2)$

$exc(e_1 - e_2) = (|e_1 - e_2| > maxint) \lor exc(e_1) \lor exc(e_2)$

$exc(e_1 \times e_2) = (|e_1 \times e_2| > maxint) \lor exc(e_1) \lor exc(e_2)$

$exc(e_1 / e_2) = (e_2 = 0) \lor (|e_1 / e_2| > maxint) \lor exc(e_1) \lor exc(e_2)$

$exc(e) = (e = nil)$ if $e$ is an integer variable.

$exc(e) = false$ if $e$ is an integer constant.
In the above version integer overflow and zero divide are detected as an exception. Given a definition of \( e.xc(e) \) for all expressions \( e \) including boolean expressions I define the mini-language \( EEPLW \) (expression exception PLW). For any program you need to know what kind of expression exceptions are being checked before you define the function \( exc \).

The mini-language syntax is the same as for \( EPLW \) and is not repeated here. The only thing that needs redefining is the first set of definitions in the configuration.

**Definition 6.6.1** The transition relation on configurations for the mini-language \( EEPLW \) consists of the following:

1. For states \( \sigma \) where \( \sigma[e.xcode] = \text{nil} \) we have the following:

   a. \( (x := e.\sigma) \to (\lambda.\sigma[\text{assign}\text{ment}[x.xcode]]) \) if \( \sigma[exc(e)] = \text{true} \)

   \( (x := e.\sigma) \to (\lambda.\sigma[e.\sigma]) \) if \( \sigma[exc(e)] = \text{false} \)

   b. \( (\text{while } B \text{ do } S \text{ end}.\sigma) \to (\lambda.\sigma[\langle \text{while } x.xcode \rangle]) \) if \( \sigma[exc(B)] = \text{true} \)

   \( (\text{while } B \text{ do } S \text{ end}.\sigma) \to (S; \text{while } B \text{ do } S \text{ end}.\sigma) \) if \( \sigma[B] = \text{true} \)

   \( (\text{while } B \text{ do } S \text{ end}.\sigma) \to (\lambda.\sigma) \) if \( \sigma[B] = \text{false} \)

   c. \( (if \ B \ then \ S_1 \ else \ S_2 \ end:.\sigma) \to (\lambda.\sigma[\langle \text{if } x.xcode \rangle]) \) if \( \sigma[exc(B)] = \text{true} \)

   \( (if \ B \ then \ S_1 \ else \ S_2 \ end:.\sigma) \to (S_1.\sigma) \) if \( \sigma[B] = \text{true} \)

   \( (if \ B \ then \ S_1 \ else \ S_2 \ end:.\sigma) \to (S_2.\sigma) \) if \( \sigma[B] = \text{false} \)
d. The transition relation members corresponding to the other constructs are the same as for EPLW.

2. For states $\sigma$ where $\sigma[ex:code] \neq \text{nil}$ it is the same as for EPLW.

3. Same as for EPLW

$\square$

Except for (ASSN), (WHILE), and (IF) all the rules and axioms of EPLW are valid in EEPLW. The following three axioms are introduced into EEPLW.

(ASSN.X)

\[
\{(\text{exc}(e) \lor Ex) \land P : (P^e_{\sigma})\}x := e\{P\}
\]

(WHILE.X)

\[
\begin{align*}
\{P \land B \land \neg(Ex \lor \text{exc}(B))\} & \quad \text{S}\{P\} \\
\{P\}\text{while}\ B \text{ do } S \text{ end } \{P \land \neg B \lor Ex\}
\end{align*}
\]

(IF.X)

\[
\begin{align*}
\{P \land B\} & \text{S}_1\{Q\} \land \{P \land \neg B\} \text{S}_2\{Q\} \\
\{Ex \lor \text{exc}(B)\} & \quad \text{if}\ B \text{ then } \text{S}_1 \text{ else } \text{S}_2 \text{ end } \{Q\}
\end{align*}
\]

Here is a simple example using the axiom (ASSN.X). Let's take the program $x := a/b$. Let us further assume that zero divide is the only kind of exception checking in the given programming language. Given that $\text{exc}(a/b) = (b = 0)$ we can conclude that the following is a valid specification for the assignment:

\[
\{(b = 0 \lor Ex) \land (x = 6) : (a/b = 6)\}x := a/b\{x = 6\}
\]
Theorem: The axioms associated with EEPLW are valid specifications and the deductive rules associated with EELPW preserve the validity of the specifications.

Proof: The proof is similar to that of EPLW.

(ASSN.X) Here we prove by cases. Case 1: Assume that $\sigma \models (P_e^r \land Z)$. In this case $\sigma[e; \text{encode}] = \text{nil} \text{ and } \sigma[e; \text{exc}(e)] = false$ and therefore the only case left is $(x := e, \sigma) \rightarrow (\lambda, \sigma[e[x]])$. It is easily proven that $(\sigma \models P_e^r)$ $\equiv$ $(\sigma[e[x]] \models P)$ and therefore $(P_e^r \land Z)x := e\{P\}$. Case 2: Similarly we can prove $(P \land \neg Z)x := e\{P\}$. With both cases proven we may conclude that the axiom is sound.

(WHILE.X) The proof is similar to the proof of (WHILE), in EPLW, and is therefore omitted

(IF.X) This is similar to proof of (IF), in EPLW, and is therefore omitted.

Since the semantics of the other constructs are the same as for EPLW they have already been proven.

☐

Section 7 Termination

Termination in EPLW is similar to termination in PLW in that you are using well-founded sets in the same manner. In EPLW every program that throws an exception without catching it is terminating program. A piece of code within a try block that throws an exception is in fact a terminating program.
For termination there is not much change in the axioms. For every axiom and deductive rule except (WHILE) you can replace \{\ldots\} with \langle\ldots\rangle to get a correct deductive rule or axiom for total correctness. For termination of a while loop you can use well-founded sets in the same manner as in \textit{PLW} except that the terminating condition is different.

\textbf{(WHILE.TERM)} Let \{I(w)|w \in W\} be a set of predicates where \(W\) is a well-founded set.

\[
\forall w_1 \exists w_2((Ex?false : (I(w_1) \land B))s(I(w_2)) \land w_2 < w_1) \Rightarrow \langle w(I(w)) while B do S end \exists y, z(Ex?I(y) : (I(z) \land \neg B))\rangle
\]

\textbf{Proof:} Assume \(\forall w_1 \exists w_2((Ex?false : (I(w_1) \land B))s(I(w_2)) \land w_2 < w_1)\)

Prove \(\langle w(I(w)) while B do S end \exists y, z(Ex?I(y) : (I(z) \land \neg B))\rangle\)

Let us start by assuming nontermination and lead to a contradiction.

1. Let \(\sigma_0, \sigma_2, \sigma_3\ldots\) be a sequence of states such that

\[
\langle while B do S end, \sigma_0 \rangle \rightarrow \langle S; while B do S end, \sigma_n \rangle \land (S, \sigma_n) \stackrel{*}{\rightarrow} (\lambda, \sigma_{n+1})
\]

and since we are assuming nontermination we have

\[
\forall i \exists w(\sigma_i \models Ex?false : (I(w) \land B)).
\]

2. We know \(\sigma_i \models I(w)\) for a given \(w\). We also know \(\sigma_i \models \neg Ex \land B\). This satisfies the precondition of the antecedent and therefore there exists a \(w' \in W\) such that \(\langle S, \sigma_i \rangle \stackrel{*}{\rightarrow} (\lambda, \sigma_{i+1}), \sigma_{i+1} \models I(w')\) and furthermore \(w' < w\).

3. From 2 we know there exists a sequence \(w_0, w_1, w_2\ldots\) such that for all \(i\)

\[
\sigma_i \models I(w_i), \langle S, \sigma_i \rangle \stackrel{*}{\rightarrow} (\lambda, \sigma_{i+1}), \sigma_{i+1} \models I(w_{i+1}), \text{ and } w_{i+1} < w_i.
\]
4. From 3 we know $w_0 > w_1 > w_2...$ and since each of these elements belong to $W$ this contradicts $W$ being a well-founded set. Therefore we have proven: \( \exists w(I(w)))while\ B\ do\ S\ end(true) \)

We know that the loop terminates. Let $P = \exists w(I(w))$.

1. \( \forall w_1 \exists w_2(\{Ex?false : (I(w_1) \land B)\} S\{I(w_2)\} \land w_2 < w_1\) (original assumption)

   (total correctness implies partial correctness)

2. \( \{Ex?false : (P \land B)\} S\{P\} \)

3. \( \{P\}while\ B\ do\ S\ end\{Ex?P : (P \land \neg B)\} \) (WHILE)

4. \( \{\exists w(I(w))\}while\ B\ do\ S\ end\{\exists y, z(Ex?I(y) : (I(z) \land \neg B))\} \) (def of $P$)

5. \( \langle \exists w(I(w))\rangle while\ B\ do\ S\ end\{\exists y, z(Ex?I(y) : (I(z) \land \neg B))\} \) since termination has been proven.

**Section 8 Object-Oriented Extentions**

This is an attempt to show how to combine $EPLW$ with $POLYPLW$ in chapter

4. Proofs of everything appearing in this section are omitted.

1. For an $S$ construct for every

   \( \langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle \)

   in the configuration of the original language the following is in the configuration for the extension of $EPLW$
\(\langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle\) if \(\sigma[\text{encode}] = nil\)

\(\langle S, \sigma \rangle \rightarrow \langle \lambda, \sigma \rangle\) if \(\sigma[\text{encode}] \neq nil\)

2. For \(C = \text{class } T; \text{ inherit } T_i \text{ export } N^* \text{ feature } V^* R^* \text{ end}\) it is the same as for the object-oriented mini-language \(\text{POLYPLW}\) in chapter 4.

3. \(\langle N : T, \sigma \rangle \rightarrow \langle \lambda, \sigma \cup \{(N, T, nil)\}\rangle\) introducing a new variable to the program state.

For every valid deductive rule,

\[
\begin{align*}
\{P_1\} S_1 \{Q\} \\
\{P_2\} S_2 \{Q\}
\end{align*}
\]

for \(S\) constructs \(S_1\) and \(S_2\) in the language \(\text{POLYPLW}\) we have the following deductive rule in the extended \(\text{EPLW}\):

\[
\begin{align*}
\{\text{Ex}?Z : P_1\} S_1 \{Q\} \\
\{\text{Ex}?Q : P_2\} S_2 \{Q\}
\end{align*}
\]

where \(Z\) is a dummy variable (you can replace \(Z\) with \textit{false}).

For every valid axiom \(\{P_1\} S \{P_2\}\) in \(\text{POLYPLW}\) we have the axiom
\(\{\text{Ex}?Q : P_1\} S \{\text{Ex}?Q : P_2\}\) in the extended language.

We explicitly state the new deductive rules corresponding to (OBJ-FUNC) and (FUNC) and leave the definitions of the other axioms to the readers.
(OBJ-FUNC):

For \( T :: \text{proc } N(a_1 : T_1 \ldots a_n : T_n) : T_r \text{ V } S \text{ end; and a new identifier } ls \) we have:

\[
\frac{\{\text{Ex}?Z : (P \land isvar(ls, V))\} S^\text{current}_x \text{ local } \{Q^y_{ls, \text{Result}}\}}{\{\text{Ex}?Q : \left[ (P_{r_1}^{ls,a_1 \ldots ls,a_n})^{ls,*}_{\text{nil}} \land (x.\text{type} = T) \right] \} y := x.N(e_1 \ldots e_n)\{Q\}}
\]

for a new identifier \( ls \) where \( ls^- \in \text{var}(Q) \)

(FUNC): For \( \text{proc } N(a_1 : T_1 \ldots a_n : T_n) : T_r \text{ V } S \text{ end;}

\[
\frac{\{\text{Ex}?Z : (P \land isvar(ls, V))\} S^\text{local}_x \{Q^y_{\text{Result}}\}}{\{\text{Ex}?Q : (P^{ls,a_1 \ldots ls,a_n})^{ls,*}_{\text{nil}} \} y := N(e_1 \ldots e_n)\{Q\}}
\]

for a new identifier \( ls \) where \( ls^- \in \text{var}(Q) \)

For type checking in \textit{POLYPLW} we can place an assert statement immediately before every object copying operation. Before the operation \( y := x \) place the following assertion:

\[
\text{assert}(\text{copy}) \ x.\text{type} \in \text{descendents}(\text{type}(y)) \text{ end}
\]

We can further define the following equivalence for any operation \( S \) where \( P \) is the condition to be satisfied for \( S \) to be successful and \( c \) is the exception code thrown if \( S \) is unsuccessful.

\[
S_{\text{extended-\textit{EPLW}}} \equiv (\text{assert}(c) \ P \text{ end; } S)_{\textit{EPLW}}
\]

This section is intended as a starting point in the combination of the two mini-languages. It is also intended for showing how a mini-language can be extended with exception handling constructs.
Section 9 Non-determinism and Exceptions

Let us consider the situation where a machine runs out of memory. Any explicit or implicit request for memory will cause an exception. It can be determined from the source code when there will be such a request, however, it cannot be determined, from the source code, whether or not there is sufficient memory to allocate. We therefore have non-determinism. An operation being nondeterministic basically means that the outcome of the operation is not uniquely determined by the state of the program before the operation. A more precise definition appears below.

Definition 6.9.1 A segment of code $S$ is non-deterministic if and only if there exist state $\sigma$, $\sigma_1$ and $\sigma_2$ such that $\langle S, \sigma \rangle \xrightarrow{*} \langle \lambda, \sigma_1 \rangle$, $\langle S, \sigma \rangle \xrightarrow{*} \langle \lambda, \sigma_2 \rangle$, and $\sigma_1 \neq \sigma_2$.

From the transition relation of $EPLW$ it is easy to see that a canthrow operation is nondeterministic. A canthrow operation can be placed immediately before every operation which may, or may not, cause an exception.

Let us consider the dynamic allocation of memory. To make sure a program never tries to dereference a null pointer we can throw an exception for every unsuccessful attempt to allocate memory. Here we consider this situation in pseudo-code (left) and its simulation using canthrow (right). (A cautionary note that new node is not in $EPLW$)
ptr:=new node;
if (ptr = nil) then
    throw(memory);
end:

Suppose the exception doesn’t occur until an attempt to dereference a null pointer. We can simulate the allocation of memory by causing a temporary exception in the event that there is insufficient memory. This means that if there is no memory left to allocate then the operation is simply not performed.

In the first program below, \(ptr2\) is a temporary variable which holds the old value of \(ptr\) so that it can be restored. In the second program below, the value of \(ptr\) will be \texttt{nil} upon a memory exception.

```plaintext
try
    canthrow(memory);
    ptr2:=ptr;
    ptr:= new node;
catch(e)
    ptr:=ptr2;
end;

try
    canthrow(memory);
    ptr:= new node;
catch(e)
    ptr:=nil;
end;
```

We can simulate the dereferencing of the pointer \(ptr\) with the following code:

```plaintext
assert(segmentation_fault) ptr!=nil end;
printf(ptr->data);
```

Every procedure can implicitly cause a memory exception. It is possible that
there is not enough memory available to allocate for the local variables. It is also possible that a program detects the condition where the machine is low on memory and causes an exception within a program as a warning signal. This can be simulated in EPLW by inserting a canthrow statement within every procedure at its beginning as illustrated below.

```epl
proc some Routine(a, b, c)
  local X, Y, Z
do
  canthrow(memory);
  the_code_for_the_routine;
end;
```

On a final note any operation which is non-deterministic in whether or not it causes an exception, can be simulated by inserting a canthrow operation before it. The name of the exception code should reflect the exception as usual.

**Section 10 Exception handling in real languages**

In this section I attempt to simulate, in EPLW, some of the exceptions and exception handling found in real languages.

The language of Eiffel supports assertions and assertion monitoring. In the event that such an assertion is violated an exception will be produced. To convert Eiffel code, with preconditions, postconditions and check instructions, all you have to do is convert each into an assert statement. A loop invariant is slightly different. If condition $I$ is an invariant of the loop `while $B$ do $S$ end;` then the
appropriate conversion is

\textbf{assert}(id) \textbf{I} \textbf{end}; \textbf{while} B \textbf{do} (S; \textbf{assert}(id) \textbf{I} \textbf{end}) \textbf{end};

Let us consider the abstract syntax \textbf{while} B \textbf{variant}(v) \textbf{do} S \textbf{end} where \( v \) is a loop variant which is not altered by the program segment \( S \). Eiffel uses the non-negative integers as the well-founded set. We therefore have the program

\[ v := e; \textbf{while} B \textbf{do} (S; \textbf{assert}(!v)) 0 \leq e < v \textbf{end}; v := e \textbf{end} \]

Take note that the translated program is guaranteed to terminate if \( S \) is guaranteed to terminate.

In Eiffel you are allowed to retry a routine whenever an exception occurs. In the event that a routine is not retried then it fails. A retrievable piece of code is defined as:

\textbf{retrievable} \( S_1 \) \textbf{catch}(e) \( S_2 \) \textbf{end}

This code is defined in terms of the \textbf{try}/\textbf{catch} constructs of C++. In order to retry a piece of \textit{EPLW} code in the same way as an Eiffel routine can be retried we need to place a \textbf{try} construct within a \textbf{while} loop. A new variable is introduced which does not appear anywhere in the program. We choose to call it \textit{retrievable}, however, you must choose a different name for this variable for every \textbf{try} block within a set of nested retrievable \textbf{try} blocks otherwise they will interfere with each other. We start with the Eiffel routine we wish to translate.

\textit{eiffelroutine} \textbf{is}
do
  \textit{S}_1
  \textit{rescue}
  \textit{S}_2
  \textit{end};

Here is the same program in \textit{EPLW}.

\textit{proc ciffer}\textit{outine}

\textit{do}

  \textit{retryable} := \textit{true};

  \textit{while retryable do}

    \textit{try}

    \textit{S}_1:

    \textit{retryable} := \textit{false};

    \textit{catch(e)}

    \textit{retryable} := \textit{false};

    \left(\textit{(S}_2\textit{retryable=true})\right):

    \textit{if} \neg\textit{retryable} \textit{then}

      \textit{throw(e)}

    \textit{end};

\textit{end};
Another approach to this problem would be to define the semantics and another proof rule for a retriable piece of code. This would be another source of nontermination as it is possible to retry a fragment of code infinitely many times.

In C++ a programmer is allowed to define several catch constructs with each construct handling a different type of exception. This can be simulated in EPLW with one or more if constructs within a single catch statement.

The C++ version

```c++
try {
    somecode
} catch(zerodivide)
    handle_zerodivide();
} catch(integer_overflow)
    handle_integer_overflow();
} catch(memory_shortage)
    handle_memory_shortage();
} catch(...)
    handle_default();
}
```

The EPLW version:

```plaintext
try somecode
catch(proc(c)
    if c=zerodivide then
        handle_zerodivide();
    else
        if c=integer_overflow then
            handle_integer_overflow();
        else
            if c=memory_shortage then
                handle_memory_shortage();
            else
```
In some languages exceptions may be handled by a separate routine then continue at the same point in the flow of control. This is easily simulated in EPLW. There is to be a try construct for every operation that is capable of causing an exception. Consider the following code:

```plaintext
assert(zerodivide) not b=0 end;
c:=a/b;
```

In EPLW if this assertion fails then control effectively goes to the next available catch statement then continues execution after the catch statement. The code below places a catch construct immediately following the operation so that the program can effectively continue where it left off.

```plaintext
try
  assert(zerodivide) not b=0 end;
c:=a/b;
catch(k)
  handle_zerodivide(a,b,c);
end;
```

Section 11 A Final Note

It is possible to verify programs with exception handlers. A mini-language called EPLW was developed with exception handling routines similar to what you would see in C++. Several kinds of exceptions and exception handling are
covered in the sections following. Several examples of routines with the different constructs are given.

It is possible to verify programs that cause exceptions with every attempt to evaluate an invalid expression. Some of the semantics of $EPLW$ were redefined under the assumption that an evaluation of an invalid while condition, if condition or right side of an assignment will cause an exception. This newer mini-language was entitled $EEPLW$ to distinguish it from $EPLW$.

Nondeterminism can also occur in exception handling. There are some situations for which it cannot be determined whether or not an exception will occur with the next operation to perform. This type of situation can be simulated in $EPLW$ using the nondeterministic pseudo-construct called $canthrow$.

Normally executable assertions are included in real languages to facilitate debugging. Executable assertions included in a mini-language intended for verification is one step towards being able to verify that a given executable assertion in a real program is always satisfied. This would be done by verifying that a program, with run time assertions, never produces an exception.

In considering the verification of programs where exceptions occur a program verification expert should not expect to simply add on to the previous deductive rules and axioms then continue working. Some of Hoare’s axiom’s will necessarily become invalid in the presence of exceptions. This is because the semantics
are different.
Chapter 7 CONVERSION

Section 1 Introduction

In this chapter we describe techniques for converting code into the mini-language POLYPLW. It is practically impossible to convert every possible programming language feature, however, an actual program becomes verifiable whenever a successful conversion is done.

Whenever we define a complete conversion routine from one language to an already defined language we are implicitly defining the new language. If there exists two languages $L_1$ and $L_2$ and we want to convert programs from $L_1$ to $L_2$ we can use a function $CV:L_1 \rightarrow L_2$ which is defined for every member of $L_1$. The conversion should preserve the meaning of the program. In other words $M_{L_1}(P)(\sigma)=M_{L_2}(CV(P))(\sigma)$ should hold for every program $P$ in $L_1$ and every state $\sigma$.

Section 2 Generic Classes

Given that ordinary classes are verifiable generic classes can just as easily be verified. A reminder that generic classes are classes with one or more parameters representing arbitrary data types. Such a class can be verified as if the generic parameters are actual data types. These data types, however, have no properties that are relevant to the correctness of the generic class.
If one wants to define the meaning of an object-oriented program with

generic classes, one option is to look upon a generic class not as a class

but as a template that generates classes. In this scheme for every instance

of a type name CNAME[GP] in the program a class is generated by taking

the source text of the generic class and substituting GP for every occurrence

of the generic parameter in the class. For more complex type names like

LINKED_LIST[ARRAY[INTEGER]] a class is generated corresponding to each

of the types LINKED_LIST[ARRAY[INTEGER]] and ARRAY[INTEGER].

A few definitions are given then an algorithm is stated which converts

programs with generic classes into programs without generic classes. This will

be very useful in defining the meaning of generic software. The algorithm make

is to be invoked on every type name with generic parameters. The algorithm, at

each step, will take a data type name appearing outside any generic class.

1. \text{arity}(t) = \text{number of parameters in } t.
2. \text{arg}(n,t) = n^{th} \text{ argument of } t

   a. let \text{arg}(0,c) be the name on the outside of the square brackets

3. \text{code}(s) = \text{code for class named } s.
4. \text{name}(c) = \text{name of class } c \text{ (including the generic parameters if any)}
5. \text{subs}(c,a,b) — substitute } b \text{ for every occurrence of } a \text{ in } c.
6. \text{new\_name}(c,k) — change the name of the class } c \text{ to } k.
type_set: DICTIONARY [TYPE_NAME, CODE];
final_code: CODE;
procedure make(t: TYPE_NAME, r: CODE) is
  local
    i: INTEGER; c: CODE;
  do
    if type_set.exists(t) then
      r := type_set.item(t);
    else
      r := code(arg(0, t));
      for i = 1 to arity(t) do
        make(arg(i, t), c);
        subs(r, arg(i, name(r)), name(c));
      end;
      new_name(r, generate_name());
      type_set.put(r, t);
      Replace every occurrence of t in final_code with name(r)
      final_code.include(r);
    end;
  end;
end;
Let us consider the following generic class:

class PAIR[A,B] export
  first,second,make_first,make_second
feature
  Create(a:A,b:B) is
    do
      first := a;
      second := b;
    end;
  first:A;
  second:B;
  make_first(a:A) is
    do
      first := a;
    end;
  make_second(b:B) is
    do
      second := b;
    end;
end;

With the declarations \( ggg: \text{PAIR[INTEGER, STRING]} \) and \( kkk: \text{PAIR[REAL, FLOAT]} \); the following is the appropriate code to include in the new source text.

class NAME1 export
  first,second,make_first,make_second
feature
  Create(a:INTEGER,b:STRING) is
    do
      first := a;
      second := b;
    end;
  first:INTEGER;
  second:STRING;
  make_first(a:INTEGER) is
    do
      first := a;
    end;
  make_second(b:STRING) is
    do
      second := b;
    end;
end;

class NAME2 export
  first,second,make_first,make_second
feature
  Create(a:REAL,b:FLOAT) is
    do
      first := a;
      second := b;
    end;
  first:REAL;
  second:FLOAT;
  make_first(a:REAL) is
```
do
  first := a;
end;
make_second(b:FLOAT) is
do
  second := b;
end;
end;
```

Also every occurrence of `PAIR[INTEGER, STRING]` in the program is replaced with `NAME1` and every occurrence of `PAIR[REAL, FLOAT]` in the program is replaced with `NAME2`.

**Section 3 “Once” routines**

In translating a once routine it is useful to introduce a new boolean attribute, denoted by `rf`, to the class. The identifier `rf` must not appear in the old class. The initial value of the attribute is false. This new attribute is to be initialized to false by the creation routine.

The routine body should be placed in an if construct so that it can be conditionally executed. The attribute `rf` having the value `true` will prevent the body of the routine from being executed.

```
From:   rname is once S end;
To:     rf:BOOLEAN;
         rname is do if (not rf) then S; rf:=true; else skip end; end;
```

For once functions there is one slight difference. Not only is a new boolean value needed but also an new variable with the same type as the return value. The
returned value from the first invocation must be stored so that it can be returned in subsequent invocations. It is assumed here that Result is the variable holding the value to be returned by the function.

From: rname:RTYPE is once S end;

To:

rf:BOOLEAN; rval:RTYPE;
rname:RTYPE in
  do if rf then Result := rval;
  else S; rf:=true; rval:=Result; end;

Section 4 Constructs within routines

The loop in Eiffel is one step short of being a while loop. The if statement is much like in other languages with zero or more elsif clauses. The following is an attribute grammar for converting ordinary eiffel code into the mini-language shown in the previous chapter.

\[ S_0 ::= \text{from } S_1 \text{ until } B \text{ loop } S_2 \text{ end}; \]

\[ S\text{.output} = S_1\text{.output}; \text{while } B \text{ do } S_2\text{.output end}; \]

| if \( B \) then \( S_1 \) \( K \)

\[ S_0\text{.output} = \text{if } B \text{ then } S_1\text{.output else } K\text{.output end}; \]

\[ K_0 ::= \text{else } S \text{ end}; \]

\[ K_0\text{.output} = S\text{.output} \]

| elsif \( B \) then \( S \) \( K_1 \)

118
\[ K_0.\text{output} = \text{if } B \text{ then } S.\text{output} \text{ else } K_1.\text{output} \text{ end; } \]

| end:

\[ K_0.\text{output} = \text{skip} \]

All case constructs can be translated into an \textit{if-then-else-elsif} construct then stated in terms of a simple if statement.

**Section 5 Arrays**

The representation of arrays is simple. Here we convert to an unconventional notation for the assigning of an element to an array. The conventional notation is good enough for accessing but not changing a specific element in the array. Let \( A(N) \) denote the value of array \( A \) at position \( N \). The following notation is also introduced:

\[ A(N, E)(M) = \begin{cases} E & \text{if } M = N \\ A(M) & \text{otherwise} \end{cases} \]

Translate \( A[N] \), occurring in any expression, to \( A(N) \). Next translate \( A[N] := E; \) into \( A := A(N, E); \). Now use the assignment axiom as defined before.

We assume here that \( A \) is a two way infinite array. Therefore a variable cannot be out of range. Specific assertions should be placed in your program if you wish to guarantee that your index is in the proper range.
Section 6 Linked Lists

Linked lists can be simulated using an array. If you have a variable whose type is a pointer to another data type $B$ then simply define an array of type $B$ (i.e. $dta : ARRAY[B]$:) and the pointers data type can be integer. The integer value indicates the position in the array and thus points to the data. Let $dtt : INTEGER$ be a new variable associated with the data type which keeps tract of which nodes are in use. To define a new node and have variable $P$ point to it simply write down $P := dtt; dtt := dtt + 1;$. Next all dereferencing like $*P$ is changed to $dta(P)$:. Now that you have an array follow the instructions in the previous section.
Chapter 8 FUTURE WORK

In this chapter we discuss several possible research topics on the subjects covered in this thesis. There are three main chapters in this thesis which contain a significant amount of material. There are several issues that were only briefly covered. This does not necessarily mean that these issues are unimportant. It simply means that they can be covered more extensively.

Other than considering polymorphism in a more complete environment I do not see much room for expansion. Polymorphism could be covered more in the area of class specifications and exception handling.

Object Pointers were covered briefly in the first of the three main chapters. Some axioms were suggested which aid in the proof of object-oriented programs with pointers to objects. What could be done with this is to concentrate on class specifications in an environment with aliasing of objects. We now have to consider situations where an object inadvertently sends a message to itself indirectly while it is processing another message. In general we cannot predict what kind of message will be sent back to an object and thus the results are unpredictable unless we are proving an entire object-oriented program as one unit.

The chapter on class specifications can be expanded to consider more features of object-oriented programs. Whether or not it can be expanded to consider the
aliasing of objects remains to be seen. It is not necessary to consider object-
pointers and exception handling since the two environments can be combined.

The language of $EPLW$ is restricted to the termination model of $C++$ meaning
that, in the event of an exception, you generally cannot retry the code in the try
block of a routine. The semantics of $EPLW$ can be modified to prove programs
where a $retry$ instruction can be invoked in an exception handler which causes
control to jump right back to the beginning of the try block.

A complete program verification environment would probably have to cover
polymorphism, aliasing, exception handling, and concurrency along with many
other features.
Chapter 9 CONCLUSION

The verification of object-oriented programs in a polymorphic environment was done through an explicit distinction between the type of a variable and the type of the object which is assigned to it. It is a notation-wise distinction and a semantic distinction.

The main reason for covering class specifications in this thesis is to create an environment where properties of classes can be found which cannot be destroyed or changed by the presence of polymorphism. Reverifying a module every time we use it in another application is somewhat akin to rewriting the module itself.

A proof system for the verification of programs with exception handling constructs was developed. The constructs are based on the try/catch construct of C++. An assertion statement and a non-deterministic construct are included in the mini-language.

Whenever possible the deductive rules and axioms should be mechanical, meaning that it should be obvious what the next step in the verification process is. The deductive rules of EPLW were developed with that concept in mind. A uniform notation Ex?P:Q for preconditions and postconditions is used.

The languages of EPLW and POLYPLW can be combined to produce an object-oriented mini-language with exception handling constructs. In this kind of situation run-time type checking can be simulated. Also situations like an
exception occurring in the middle of an object being created or in the middle of an routine call on an object are easily simulated.

A chapter is dedicated to the conversion from real languages to the object-oriented mini-language POLYPLW in this thesis. Such things as generic classes (sometimes called template classes) are easily verified after an appropriate conversion.

The correctness of software should not be considered an optional luxury. With a sufficient amount of effort, writing defect-free software can be done. Every programmer is capable of making errors, which justifies the practice of “debugging”. Testing a piece of software does not guarantee its correctness, therefore, program verification is justified.

A library of reusable software components can be constructed with the capability of storing specifications along with each component. A proof of the correctness of each software component can be stored, for future reference, along with the software component itself. This will result in faster verification of new programs that use components of this library.

Exception handling is done for the purpose of taking care of unexpected events, however, if exceptions are handled improperly then the exception handling may cause more problems than it solves. This justifies the verification of programs with exception handling.

124
The development of formal methods for the verification of computer programs is a necessary step in the development of (semi-)automatic program verifiers and computer aided program verification. The rules must be sound to be of any use. Rules for more complex programming language constructs can usually be developed with the aid of existing deductive rules for simpler constructs. The assignment axiom $ASSN$ from $PLW$, for example, can be used to develop rules for more complicated assignments.
Computer Program Verification (A survey)

Abstract

I shall review computer program verification in terms of the methods used and the common programming language features. This survey is confined to deterministic programs. Automatic verification is valuable since it relieves the programmer of the extra work. Some existing prototypes and methods for automatic verification are briefly explained. All the major models of computer program verification are covered in this report. Hoare's axiomatic approach appears to be a model for most of the techniques on program verification despite the fact that termination is not covered in the original paper. Another model is Floyd's method.

Section INTRODUCTION

Because the development of computer programs is an error prone process, there is a need for methods of validating programs. The traditional method of validating a program is to test its performance on a finite, and usually small, number of cases. The method is nicknamed "debugging". As early as the 1940s when programs were becoming more and more complex, people were unsatisfied with this method. This is no doubt part of the reason program verification has become an important field of computer science.
The practice of "debugging" does not guarantee the correctness of software as it doesn't cover all cases. A mathematical approach is more appropriate for validating software. The discipline of program verification is such a method. Program verification is the practice of proving, through formal methods, that a program will always give the correct answer (under the assumption that the underlying hardware and implementation of the programming language is totally reliable).

The Problem

The basic challenge after writing a piece of software is to prove it. The best way to validate a program, if it can be done practically, is to do a mathematical proof of the program. Program verification studies the formal approaches for proving software.

Overview of Approaches

Some approaches have been established for program verification. They include intermittent assertions, strong verification, structural induction, and subgoal induction. Structural induction is induction on the attributes of the data being operated on. Subgoal induction is a method which breaks down the problem into smaller problems, solves them, then uses those solutions to solve the original problem.
Organization of The Paper

I will start by assessing the role of program verification in industry which is an important issue. In Section 2 I describe a basis of program verification with material from milestone papers and other important papers. Section 3 studies program verification in terms of the verification methods used. Section 4 describes program verification in terms of the programming language features to be verified. The subject of the Section 5 is automatic program verification. Some miscellaneous concepts related to program verification are discussed in Section 6. Notations are summarized in section 7. Final comments appear in section 8.

Section 1 The role of program verification

Program verification is important since the correctness of a program dictates the usefulness of a program in the first place. Critical comments about the usefulness of program verification in industrial programming can be found in
[25] and [29]. The arguments of [25] are summarized as follows:

It is a social process that determines whether mathematicians feel confident about a theorem. Since such a social process can never take place among program verifiers the discipline is bound to fail. Formal verifications of programs will not play the same key role in computer science and software engineering as proofs do in mathematics. Since change is unavoidable and specifications of programs are complex the role of formal verification is very difficult to justify. Algorithms as logical structures, are appropriate for deductive reasoning but programs are not.

The communications of the ACM issues of March-April 1989 contain arguments against and in favor of the application of program verification to real programs. A critical review of this debate is presented in [7] centered around the article by James Fetzer [29] and is intended to draw the readers attention to the debate surrounding program verification. The author of [29] believes that program verification will never be completely successful and is therefore useless.

A whole issue of the ACM SIGSOFT software engineering notes, Volume 5, Number 3, July 1980, presents the minutes of a workshop on program verification, with many positive accounts of application.

The article [15], by Boyer and Moore, investigates many approaches to program correctness which are united by the view that programming is a mathematical
activity. There are different views concerning the validation of programs: Should one strive for formality? Should proofs be mechanically checked? Are new programming languages required? Should new logics be used?

[12] is a position statement for a workshop for which the purpose was to provide a forum for those who have used formal methods in an industrial setting to discuss their experience. Also discussed is how formal methods are being applied in an industrial production environment. This position statement is an attempt to define a system of coordinate systems within which one might discuss the proper role of and experience with formal methods in software development.

FM89 [22] was an invitational workshop bringing together representatives from the research, commercial and governmental spheres of Canada, the United Kingdom and the United States. The primary purpose of the workshop was the assessment of the role of formal methods in the development of critical software. Critical software includes any software that may be a threat to public safety upon failure to perform correctly. An example would be a system for controlling a nuclear power plant.

Section 2 A basis for program verification

Before Hoare's axiomatic method

A snapshot [72] refers to one particular moment in the development of the process applied to some data. A general snapshot is a snapshot of a dynamic
process which is associated with a particular point in the program text. The method of general snapshots is not a set of specific rules but rather a view of programs which can be used in proofs.

Floyd's basis

According to [30], a milestone article by R. W. Floyd, the basis of program verification is to associate a proposition with each point in the flow of control within a program. The proposition is asserted to hold whenever flow of control reaches that point. Properties of programs to be proven are usually of the form "If the initial values of the variables in the program satisfy the relation R₁ then the final values of the variables will satisfy a relation R₂". Proofs of termination are done by showing that some entity decreases with every step in a program and the same entity cannot decrease indefinitely.

A flowchart program is drawn as a directed graph with nodes representing commands and edges representing possible passages of control between commands. An interpretation of a flowchart program is a mapping of its edges to propositions. A verification of an interpretation of a flowchart is defined as a proof that for every command c if control ever reaches that command in a state where P is true, then control must leave in a state where Q is true. P and Q are propositions on edges leading into and out of node c respectively. A semantic definition of a particular set of commands is a rule for constructing, for any
commands in this set, a verification condition.

It is possible to extend a partially specified interpretation to a complete
interpretation provided that initially there is no closed loop in the flowchart with
any untagged edges or untagged entrances. Automatic program verification is
possible as a result of this.

Proofs of termination of a flowchart program can be done by using well-
ordered sets. A function $f$ mapping every edge to a member of $W$, a given
well-ordered set, is introduced. If it can be shown that with each execution of
a command the current value of $f$ is less than its prior value, it follows that the
program (or flowchart) terminates.

The author of [24] proved that Floyd’s method, which later became known
as the inductive assertions method, is complete.

**Hoare’s axiomatic method**

The article [41], a milestone paper by C. A. R. Hoare, introduces the ax-
omatic approach in defining the meanings of the different programming language
constructs for the purpose of being able to verify programs which use such con-
structs. The most important property of a program is whether it accomplishes
the intentions of its user. These intentions should be stated by making assertions
about the values of variables at the end of the execution of the program or at
several intermediate steps.
There are several axioms as tools for expressing the meaning of programs for the purpose of verification. (See page 155 for an explanation of the notation used)

The Axiom of Assignment is expressed as:

\[ P0(x:=f)P \]

where x is a variable, f is an expression, and P0 is obtained from P by substituting f for all occurrences of x. The Rules of Consequence is expressed as:

If \( P\{Q\}R \) and \( R\Rightarrow S \) then \( P\{Q\}S \) and \nIf \( P\{Q\}R \) and \( S\Rightarrow P \) then \( S\{Q\}R \).

The Rule of Composition is expressed as

If \( P\{Q1\}R1 \) and \( R1\{Q2\}R \) then \( P\{Q1;Q2\}R \).

The Rule of Iteration is expressed as

If \( P \& B \{S\} P \) then \( P \{\text{while } B \text{ do } S\} \; \beta \& P \)

The axioms and rules of inference, stated above and quoted in [41], implicitly assume the absence of side effects, of the evaluation of expressions and conditions. In proving properties of programs expressed in a language permitting side effects it would be necessary to prove the absence of such side effects before applying the appropriate proof technique with Hoare's axioms. Another deficiency of the method is the inability to prove that a program terminates. Therefore the axioms
can only be used to prove that "if a program terminates then it will give the correct result".

**A theorem of loop assertions**

Another contribution to program verification is the *Fundamental theorem of loop assertion* stated in [8]. Let $W(B,S)$ represent the code segment *while B do S end*, and $D$ be a subset of the set of n-tuples which when input to $W(B,S)$ lead to termination. $P$ is an invariant of loop $W(B,S)$ if and only if $[(P \land B) \implies S \implies P]$.

**Fundamental Theorem of Loop assertions:** $W(B,S)$ computes $F$ over a range inclusive $D$ if and only if:

1. Input with any $x$ in $D$, $W(B,S)$ terminates with output in $D$,  
2. $[x \in D \land ^\ast B(x)] \implies [F(x) = x]$, and  
3. $\{y \in D, x \in D \implies [F(x) = F(y)]\}$ is a loop assertion for $W'(B,S)$.

The third rule says that whenever any intermediate $x$ is in $D$ then $x$ must necessarily contain all the useful information regarding output.

**Other issues**

Programs can be proven with one unified formalism as suggested by the proof rules in [63]. One can prove a program in one step or in several steps with each step establishing more properties. There is a need to establish the relation between variable values before and after execution. There are several topics one
might consider under the heading of proper termination (underflow, overflow etc). However, what is dealt with conventionally is ensuring against endless execution.

A question arises for any new proof system [19 page 78]: How can we be sure that we are not omitting some restrictions on the rules or other items that are necessary to ensure the validity of the rules? One way is to prove that all the axioms and rules are true in the interpretative model.

Section 3 Methods

This section deals with some of the different methods of program verification. Some of these methods can be combined to prove programs.

Intermittent assertions

The intended result of using intermittent assertions [64] is the ability to prove the partial correctness and termination of a program simultaneously.

The technique of intermittent assertions involves the affixing of comments within a program with the intention that “sometime” control will pass through the point and satisfy the attached assertion. The statement denoted by

\[ \text{sometime } Q \text{ at } L \]

states that flow of control will eventually pass through label L with assertion Q satisfied. This implies that control may pass through label L many times without Q being satisfied but must eventually pass through L with Q being satisfied. A couple of statements are of importance. The first is an expression of the
total correctness assertion of an algorithm, with input specification P and output specification Q.

If sometime P at start then sometime R at finish

The second statement is the assertion that the same algorithm terminates but does not state any specific result.

If sometime P at start then sometime at finish

Strong verification

As opposed to the traditional approaches to proving termination and consistency, strong verification [9] is based on the premise that the termination property of a program S is implicit in the algorithm that realizes S. The iteration axiom uses an extra predicate associated with the number of times the loop executes.

Structural induction

Structural induction [16] is a method of program verification by which a program is proved for the most elementary data. Secondly it is proved that a program works for data of an arbitrary level of complexity provided that it works for all data of lesser complexity. Any expression in a language, which is a candidate for structural induction, is either an atom or a structure. An example of such an expression is a list defined by the following:

1. An $\alpha$--list is either a cons or a nil.
2. A cons has an $\alpha$ and a list.
3. A nil has no components.

**Subgoal induction**

The technique of subgoal induction [71] consists of a proof rule satisfying the problem solving heuristic: "Transform a problem into a smaller problem with the same general characteristics, solve the smaller problem then use it to solve the original problem".

Suppose a loop is specified by the requirement that a given input \( x \) is to produce output \( z \) such that \( \psi(x;z) \) is to be satisfied. The test condition \( P(x) \) is the condition to exit from the loop. In the loop \( x \) is assigned \( N(x) \). The following rules are applicable:

\[
P(x) \Rightarrow \psi(x;x)
\]

\[
\neg P(x) \land y = N(x) \land \psi(y;z) \Rightarrow \psi(x;z)
\]

It is shown in [71] how the synthesis of a recursive function can be coupled together with the induction rule and thus a recursive program can be constructed and verified simultaneously.

Subgoal induction for recursive procedures or programs is more natural than subgoal induction for iterative programs. A forward proof can be converted into a subgoal proof [50].

**Section 4 Verification of some features**

Program verification would be useless if there were no methods of verifying
the features consistently used by programmers. This section reviews program verifycation in terms of some of these special features and attempts to show how programs with some of these features can be verified. The features appear in alphabetical order.

**Aliasing**

Aliasing occurs when two or more names represent the same memory location. This feature is allowed in most programming languages. The source of aliasing in most cases is formal parameters and procedures. Aliasing can also be an indirect result of other features of programming languages. Aliasing usually occurs when a procedure is called with an actual parameter which is the same as a global variable in the scope of the procedure body.

Aliasing is rigorously dealt with in [17]. Also [1] deals with aliasing. The paper [14] explicitly distinguishes between variables and their values allowing one to conveniently talk about aliasing of variables.

**Arrays**

The array is an important data structure. Papers [35], [45] and [58] deal with arrays. The conventional assignment axiom becomes unsound when attempting to perform certain assignments involving subscripted variables. This fact was pointed out by [31]. There is a solution to the problem.
If $A$ is an array then the proof rules

$$P^A_{assign} (A,I,E, \{ A'[i] := E \})$$

and

$$P^X_{select} (A,I, \{ X := A'[i] \})$$

are appropriate. The function $assign(A,I,E)$ represents the array $A$ with the element in its $i$th position replaced by the expression $E$. The function $select(A,I)$ represents the element in the $i$:th position of array $A$.

**Functions**

Functions are present in most programming languages and in some cases have side effects (Side effects are dealt with on page 144). Functions are basically routines that return a value. Any flowchart program can be modelled as a functional program [62].

One proof rule in [18] assumes that a function is declared as $f(x)$ function $Q$, where $f$ is the function name, $Q$ the body, and $x$ the formal parameters. The proof rule assumes that no side effects can occur within a function body. In many programming languages the function name is used to store the result of the function. Under this assumption the proof rule is stated as:

$$\forall x \left( P \Rightarrow R^f_{f(x)} \right)$$

139
There is a caveat in the use of the above rule. It is the case where the function never terminates and, therefore, is undefined. Let \( Q = f := 0; \text{while true do null} \) (example is from [4]). Using \( f=0 \) as an invariant it can be proven that \( \text{true} \{ Q \} \) \( \text{false} \& f=0 \). The rule of functionality gives the following:

\[
\begin{align*}
    f(x) & \text{ function } Q \\
    \forall x (\text{true} \Rightarrow \text{false} \& f(x) = 0)
\end{align*}
\]

The above is a contradiction in the conventional interpretation. This can be avoided by insisting that \( R \) contain \( f \) and that \( \forall x P(f(x)) \) be interpreted as \( \forall y \forall x ((x, y) \in f \Rightarrow P(y)) \) where \( y \) is a new identifier.

The article [62] deals with functional programs defined by conditional recursive expressions.

**Jumps (goto's)**

Jumps are a controversial programming language feature available in many programming languages. Jumps are basically statements which cause the flow of control to jump to another location in the program.

[18 appendix III] introduces a proof rule in Hoare's logic with respect to *goto* statements. The following notation is given: \( Q_1; l; Q_2 \), where \( l \) is the label, and \( Q_1 \) and \( Q_2 \) are simple or compound statements. It is assumed that \( l \) is the only label in the block. If \( S \) is the desired precondition to each jump to \( l \), then \( S \) must
also be true on termination of \( Q_1 \). Jumps to \( l \) may occur within \( Q_1 \) or \( Q_2 \). Here is the proof rule:

\[
S \{ \text{go to } l \} \text{false} \vdash P \{ Q_1 \} S \\
S \{ \text{go to } l \} \text{false} \vdash S \{ Q_1 \} R \\
\frac{\ }{P \{ Q_1 : l ; Q_2 \} R}
\]

This is obviously a very complex looking rule. A new notation is introduced in [81]. Instead of \( P \{ S \} Q \), it uses \([b : P \{ S \} c : R]\). The interpretation is that if the proposition \( P \) holds at some time instant when the flow of control is at the label \( b \) in \( S \), then, at some subsequent time instant, flow of control will reach the label \( c \) with \( R \) satisfied. Moreover, during the intervening time, flow of control stays in \( S \) except during the executions of procedure invoked from \( S \). With \text{in} indicating the starting statement of \( S \) and \text{out} indicating the exiting statement from \( S \), the statement \([\text{in} : P \{ S \} \text{out} : R]\) constitutes a total correctness assertion.

Included in [81] is a set of axioms and proof rules using this notation. This notation is more flexible than the original notation and in my opinion is more suitable for goto statements than the above rule. Multiple jumps can be accommodated much easier with this new notation than with the conventional notation. Also the associated proof rules for it are easier to understand in the context of goto statements.
Multiple exits can be dealt with as proved by [3]. It is important to note that goto statements destroy the usual conventions for program specifications, namely that each statement has one entry and one exit. The newly introduced notation is described as follows. The basic axiom for a goto statement is
\[ \forall P \{ \{ P \} \text{goto} l \{ \text{false} \} \{ L : P \} \} \] and the basic axiom for a statement \( S \) without a goto is \[ \{ \{ P \} S \{ Q \} \implies \{ P \} S \{ Q \} \{ L : \text{false} \} \} \] and the main proof rule is:

\[
\frac{\{ \{ P \} S \{ Q_1 \} \{ L_1 : R_1 \}, \{ Q_1 \} S_2 \{ Q_2 \} \{ L_2 : R_2 \}}{\{ \{ P \} \text{begin} S : S \text{end} \{ Q_2 \} \{ L_1 : R_1 \} \{ L_2 : R_2 \}}
\]

The other axioms are stated in the article. Note that the extra bracketed expressions attached to the specifications each represent a point in the flow of control which can be jumped to from within the code.

**Multiple assignment**

A multiple assignment is an assignment to more than one memory location.

Let \( \overline{x} \) be a list of simple different variables \( x_1, \ldots, x_n \) and \( \overline{e} \) be a list of expressions \( e_1, \ldots, e_n \). Then \( x_1, \ldots, x_n := e_1, \ldots, e_n \) is conventionally defined as

\[ \{ P_{x_1, \ldots, x_n} \} x_1, \ldots, x_n := e_1, \ldots, e_n \{ P \} \] or abbreviated as \( \{ P_{\overline{x}} \} \ \overline{e} := \overline{x} \{ P \} \). The definition is from [36]

Such assignments should rarely ever occur. However, multiple assignments are indispensable in talking about procedure call proof rules.
Procedure calls

Just about every programming language is procedurized. There are many papers concerning procedure call proof rules. They are [1], [11], [10], [17], [19], [28], [31], [36], [42], [66], [74], [79], and [81].

In what follows in the procedure call $S(a,b,c)$, $a$ represents the list of input parameters, $c$ the list of output parameters, and $b$ the list of input/output parameters.

A proof rule for the procedure call $S(a,b,c)$ is a method of associating with any predicate $E$ a second predicate that implies $wpt(S(a,b,c), E)$. This predicate is called the derived precondition. It must depend on the specifications alone and not the details of the construction of $S$. Let us assume that $\{U\} S \{V\}$ has been proved for procedure body $S$.

The proof rule in [66] for determining whether a call establishes predicate $E$ is as stated. An adaptation $A$ is introduced such that it is an invariant of the body of $S$.

Proof Rule: Let $A$, the adaptation, be a predicate then:

\[
\frac{\left[ A \land V \Rightarrow E_{y,c}^{h}\right] \land \left[wlp(S, A) = A\right]}{\left[(A \land U)^{x,d}_{a,b} \Rightarrow wpt(p(a,b,c), E)\right]}
\]

The paper [10] introduces another proof rule. It is claimed that it expresses the weakest precondition that can be solely inferred from the procedures speci-
fications. It is stated as:

\[(\exists m : U_{a,b}^{x,y}) \land (\forall y, z : (\forall m : U_{a,b}^{x,y} \Rightarrow V_{a}^{z}) \Rightarrow E_{y,z}^{b,c})\]

The paper [36] deals with procedure calls with aliasing. The paper [28] deals with procedures where the body of the procedure contains global variables that do not occur in the assignment position.

**Recursion**

In proving recursion most of the conventional methods for procedures are not sufficient. The solution is to attempt to prove a certain property of the body of a recursive procedure based on the assumption that all recursive calls possess the same property [42]. A similar rule would apply to mutual recursion.

In [39] a version of Hoare’s system for proving partial correctness of recursive programs is extended.

Mutual recursion can be verified. Let us assume that \( p = \{p_1, p_2, ..., p_n\} \) is a vector of procedures closed under mutual calls (procedures in \( p \) do not call anything not in \( p \) [79]. Then their properties have to be derived simultaneously with the same recursion depth counter (for termination).

**Side effects**

Side effects are a feature of programming languages. The kind of side effects I will be talking about here are expressions whose evaluation can change the state
of the program. It is useful to introduce a distinguished variable $\sigma$ representing the value returned by an expression. Thus an expression is looked upon as a command. The following axioms are from [51] and seem to be the most easily understood.

Constants: $P^\sigma_r \{ c \} P$ if $c$ is a constant. (similar for variable $x$)

Unary operators:

$$\frac{P \{ E \} Q^\sigma_{\alpha,\sigma}}{P \{ \oplus E \} Q}$$

for an arbitrary unary operator $\oplus$.

Binary operators:

$$\frac{P \{ E \} Q^\sigma_{\tau,\sigma}, Q \{ F \} R^\tau_{\tau,\sigma} \quad P \{ E \star F \} R}{P}$$

for an arbitrary binary operator $\star$.

The assignment axiom:

$$\frac{P \{ E \} Q^\sigma_{\alpha}}{P \{ \pi := E \} Q}$$

A function with more than one parameter would have a similar axiom or rule associated with it but somewhat more complex than for just unary operators. Justifications of the above axioms will be found in [51].
Section 5 Automatic verification

Automatic program verification is desirable since it reduces the amount of work to be done. There are several issues concerning automatic program verification and several prototype systems.

Axiomatization of Pascal and Euclid

An axiomatic definition, but not necessarily an implementation of a verifier, of a subset of Pascal is done in [45]. This was done so that implementors would have a rigorous definition of the language. This is, no doubt, the first step in the design of an automatic program verification system for any language.

An axiomatic definition of Euclid, a programming language intended for the expression of system programs to be verified, is found in [57]. In describing the Euclid proof rules the paper has used as much as possible the Pascal axiomatization.

Lucid

Lucid (see [5]) is both a programming language and a formal system for proving properties of Lucid programs. The properties of a Lucid program are also stated in Lucid. A Lucid program itself is simply an unordered set of assertions, or axioms, from which other assertions may be derived.

The idea in Lucid is that programs should be “denotational” and “referentially transparent” even when they contain assignment statements. This means that all
expressions in a program must mean something. Semantically, all expressions in programs without nested loops will denote infinite sequences of data objects. Reassignment to a variable must be done by using the special Lucid function next.

VISTA

VISTA (described in [34]) is a prototype system which provides assistance in synthesizing correct inductive assertions. Given only the source program it is able to generate a useful class of assertions automatically. VISTA, in 1975, consisted of a number of loosely coupled specialist modules.

VISTA uses four basic methods to obtain inductive assertions:

- Symbolic evaluation in a weak interpretation
- Combining output assertions with loop exit information to obtain trial loop assertions
- Propagating valid assertions forward through the program, modifying them as required by the program transformations
- Extracting information from failed proofs and using the information to determine how assertions should be strengthened

In many programs there is a useful amount of information that is obvious upon inspection. Weak interpretation attempts to derive simple facts of this sort. It considers only simple linear equalities or inequalities. The outcome of weak
interpretation is almost never enough to verify a program but it does provide some basic information.

Suppose a loop is exited when \( D \) is true and that outside the loop \( P \) is to hold. Therefore \( \{ D \Rightarrow P \} \) must be true inside the loop, just before the exit test. This is the weakest inductive assertion at this point since any complete inductive assertion must imply it. If it is not complete then it must be strengthened.

Whenever an assertion is known to be valid then it is useful to propagate it forward in the program, deriving the strongest consequences of the assertion downstream. Consequences of a valid predicate propagated downstream to a cutpoint may add new information which is absent in the inductive assertion at that cutpoint.

When proving mathematical theorems, if some approach fails then it is often useful to analyze the cause of the failure then fix the fault. This idea carries over to program verification.

The major components of VISTA are the weak interpreter, the trial generalization generator, the predicate propagator, and the assertion correctness and predicate initialization mechanism.

**Loop predicates**

First of all a loop predicate is a predicate which if satisfied will be true before the loop is entered and will be preserved by every loop iteration. It is possible
to automatically generate loop predicates according to [82]. Trial loop predicates which are loop inconsistent are modified according to various heuristics to generate better trial heuristics. It is also possible to accept a programmer supplied trial predicate which gives the basic idea behind a loop then fill in the details to arrive at a complete and correct loop predicate. Below are some heuristic methods summarized from [82].

1. Convert to disjunctive form and drop some of the disjuncts which come from the exit predicate.

2. To verify a predicate, convert to conjunctive form and prove each of the conjuncts separately. This may be a losing strategy if two or more conjuncts can only be proved in concert. However when it works it reduces the size of the formulas to be manipulated and hence cuts the computation.

3. If the application of the previous heuristic results in verifying some but not all of the conjuncts at some arc A then record the ones which do work. The rest of the heuristic can be seen in the article.

**VCG**

It is possible to generate verification conditions using a subgoaler for the deduction system. The subgoaler described in [46] is referred to as **VCG** (verification condition generator). VCG is intended to be the initial part of an automatic verification and debugging system. The core rules are rules from which
all other rules are derivable. A model M is assumed for the theory T of true boolean expressions of pascal and a set of programs. It is shown how M can be extended to M* for the core. The rules V used by VCG to generate subgoals are combinations of the core rules. For each procedure call the corresponding procedure is assumed to be verified. Some restrictions on V with respect to procedure calls are listed below.

1. Procedures may contain no global variables.
2. Parameters must contain distinct identifiers.
3. No actual parameter may appear in the expression list.
4. VCG does not allow a component of an array as an actual parameter.

The top level of VCG terminates because: All rules except the conditional rule generate one or two subgoals with fewer statements as they process and even the process using the conditional rule is decreasing.

Section 6 Miscellaneous concepts

Here are some miscellaneous concept concerning program verification which are arranged in alphabetical order. This includes several definitions.

Conversion

The paper [59] describes a method of converting a program P into first order formula W such that:
• The program $P$ is partially correct if and only if $W$ is satisfiable.
• The program $P$ is totally correct if and only if $\neg W$ is unsatisfiable.

This involves the conversion of a program into a sequence of statements in the form of if-then-else constructs with a single assignment and a goto statement occurring in each of the clauses.

**Decomposition Rule**

The decomposition rule [80] is designed to reduce the complexity of the theorem proving involved in program verification using Hoare's logic. The length of theorem proving is not less than exponential in the length of the theorem to be proven. Provided that $P_1$ and $P_2$ are not affected by $S$ then:

$$
\frac{\{P \land P_1\} S\{Q_1\}, \{P \land P_2\} S\{Q_2\}}{\{P\} S\{(P_1 \Rightarrow Q_1) \land (P_2 \Rightarrow Q_2)\}}
$$

This rule serves the purpose of cutting the lengths of the assertions in the proofs of $\{P \land P_1\} S\{Q_1\}$ and $\{P \land P_2\} S\{Q_2\}$ in half. The problem of how to decompose is difficult.

**Equivalence of programs**

Proving the equivalence of two programs can be valuable since the correctness of one program implies the correctness of another program. In [20] equivalence of computations and a general equivalence theorem are given.
Incorrectness proving

With all of the studies of proving programs correct what about proving that incorrect programs are in fact incorrect? In fact there is some research into proving incorrectness and proving nontermination. The benefits are that errors can be detected and eliminated. Article [49] deals with the proving of the incorrectness, and/or nontermination, of a program. It is important to note that failure to prove a program correct by some system does not necessarily imply that the program is incorrect.

It is shown in [48], through an example, that the exit approach is suitable for proving nontermination of an algorithm.

Invariant generating

Invariants can be generated through analysis of the assignment statements. Assignment statements which are on the same path through the loop must have been executed an identical number of times. Thus the counter $n$ can be used to relate the variables iterated. The method in [49] uses the above for deriving an algorithm for generating invariants.

In the same paper are some heuristic approaches. One of them is to strengthen existing invariants. Another is to weaken existing candidates.

Predicate transformers

The following definition of predicate transformers are from [66]. $wp(S,R)$ is
the weakest condition on the initial state such that execution of $S$ is guaranteed
to terminate in a state where $R$ holds. $wp(S,R)$ is the weakest condition on the
initial state such that execution of $S$ either does not terminate or terminates in a
state where $R$ holds. The properties below hold for the two functions $wp$ and $wlp$.

$$[wp(s, false) = false]$$

$$[wlp(s, true) = true]$$

$$[wp(S, R) = wlp(S, R) \land wp(S, true)]$$ for any $R$

If $B$ is a non-empty bag of predicates then:

$$[(\forall P : P \in B : wp(S, P)) \Rightarrow wp(S, (\forall P : P \in B : P))]$$

If $B$ is a set of predicates then:

$$[(\exists P : P \in B : wp(S, P)) \Rightarrow wp(S, (\exists P : P \in B : P))]$$

For all $P$ and $Q$: $[P \Rightarrow Q] \Rightarrow [wp(S, P) \Rightarrow wp(S, Q)]$

**Relational model**

In the relational model, introduced by [47], a program is represented by a set
of pairs $(s, st)$ for which execution of the program starting in state $s$ can terminate
in state $st$. The extra state $ot$ is used to deal with nontermination.

**Termination**

A program is said to terminate if for all legal input values the program
will eventually reach a halt statement (finish executing). Floyd's method which
uses the "no-infinitely-descending-chain" property of well-founded sets is the
traditional method of proving termination. Floyd's method can be generalized by showing that from every cutpoint another cutpoint will be eventually reached with a drop in the function.

In the loop approach a counter is associated with each loop reflecting number of times the loop has been executed. It must be shown that all the counters are absolutely bounded from above. An side-benefit of the loop approach is that the added information can indicate the time complexity of an algorithm.

In the exit approach, termination is shown by directly proving that for each loop the conditions for exiting the loop must be true at some point in the computation. A program will terminate if for every cutpoint either (a) such a condition, causing an exit from the loop, will eventually hold or (b) the cutpoint is never reached. One difficulty of the method is that it is often not feasible to directly show that certain values will occur during execution of the program.

In Burstall's approach it is shown that if some property $Pa$ is assumed at point $A$ then another point $B$ must eventually be reached with property $Qb$ true. This is shown by induction on the possible values of the data domain.

The paper [48] deals exclusively with termination

Well founded sets

Well founded sets have been used consistently to prove termination in most of the material dealing with termination. A definition appears in [31 page 45]
appearing below.

**Definition:** A partially ordered set \((W, \prec)\) is **well founded** if and only if there does not exist an infinite sequence \(w_i \in W, i \geq 0\), such that \(w_0 \succ w_1 \succ w_2 \succ \ldots\)

One example of well founded set is the natural numbers with their usual ordering. Another is the set of all n-tuples on the natural numbers with lexicographical ordering.

**Section 7 Summary of Notation**

There are several important notations related to program verification and used more than once throughout this survey. That is why I have included this section.

The notation \(\langle P \rangle S \langle Q \rangle\) (sometimes written as \(P \langle S \rangle Q\)) states that whenever program \(S\) starts execution in a state satisfying predicate \(P\) then if it terminates it will do so in a state satisfying predicate \(Q\).

The total correctness assertion can be stated as \(\langle P \rangle S \langle Q \rangle\) [31 page 20]. This states that whenever program \(S\) starts execution in a state satisfying \(P\) then it will terminate in a state satisfying \(Q\).

The notation \(P^\mu_b\) represents the predicate resulting from replacing every occurrence of \(a\) in predicate \(P\) with expression \(b\). If \(a\) and \(b\) are vectors then the above represents the predicate resulting from replacing every occurrence in \(P\) of a variable in \(a\) with the corresponding expression in \(b\).
The notation \([b:P(S)\rightarrow R]\) expresses that if flow of control ever reaches label 
\(b\) in \(S\) satisfying predicate \(P\) then at some subsequent point in time flow of control
will reach label \(e\) in \(S\) satisfying predicate \(R\).

Section 8 CONCLUDING COMMENTS

I included in this survey, the major models for computer program verification.
These include a paper by R. W. Floyd and another article by C. A. R. Hoare.  
Hoare's axiomatic method provides the basic axioms and notation used by a 
majority of the articles in this survey.  I also reviewed the major methods of 
program verification, and the methods for the verification of a rich variety of 
programming language features.

Automatic verification of computer programs is very convenient since it 
relieves programmers from the laborious work involved in verifying a computer 
program.  That is why I have surveyed articles discussing the automatic verification 
of programs.

I purposely left out some papers from this survey, having to do heavily with 
the verification of nondeterministic, parallel, concurrent, or distributed programs. 
I did this because there are far too many papers concerning the verification of 
such classes of programs.  Also, they are not an essential contribution to program 
verification as a distinguished discipline.
Section 9 Review of papers

In this section I have given a brief description of the contributions to the field of program verification from each paper. The descriptions are at least one or two sentences. The more important papers will usually have more description to them.

[1] In this paper operation decomposition proof obligations are given for a language with blocks and unrestricted procedure calls with reference parameters and global variables. Aliasing, as arising from reference parameters, and static scoping are dealt with in the Hoare-like proof system by the use of a "syntactic environment".

[2] This paper was written to discredit a Hoare like proof system for total correctness by showing that some of the rules can interact in such a way as to produce incorrect results. A new formalism is presented.

[3] This paper presents a program specification format which can be used for statements with multiple exits.

[4] This paper points out a serious danger in the use of the proof rules in the paper mentioned in its title. It is the case where a function never terminates and therefore is undefined. The solution proposed is an unconventional interpretation for the function notation. With this interpretation any formula containing a functional form is trivially true whenever the function is undefined.
[5] This paper describes a system for writing and proving programs in a language called LUCID.

[6] This paper provides axiomatization of a logic which reflects the possibility of undefined values for recursive and iterative definitions. An axiomatization of such logic together with examples of its usage are also provided.

[7] This paper covers the controversy surrounding program verification.

[8] Some results are extended to classes of loop programs that are not well behaved and to FOR statements. Applications of these ideas to the problem of mechanical generation of assertions is discussed.

[9] It is shown in this paper that every do-while program has a loop invariant that is both necessary and sufficient to prove strong verification. (key words: Consistency, fixpoint, loop invariant, Q-adequate, strong verification, termination, weakest precondition)

[11] The equivalence of Gries’s and Martin’s proof rule for procedure calls is established. Also presented is a modification of these proof rules for the case that the specification of a procedure contains free constants.

[10] In this paper a proof rule for the procedure call is presented. It is also claimed that the precondition it defines is the weakest precondition that can be inferred solely from the procedures specification.

[12] This is a position statement for a workshop that was for the purpose of
discussing the role of formal methods in real life applications.

[13] The program verification methods of fixed-point approach, the inductive assertion method and the weakest precondition calculus are the topics of the paper. A comparison of these methods is done.

[14] This paper is oriented around imperative languages which do not distinguish between statements and expressions. A system is presented which covers side effects and aliasing.

[15] This paper is an investigation of the many approaches to program verification.

[16] This paper discusses the technique of structural induction for proving programs correct. The paper treats programs with recursion but without assignments or jumps. An example given is a proof of the correctness of a compiler for expressions.

[17] In this paper a new version of Hoare's logic is presented that correctly handles programs with aliased variables. The paper also deals with reference parameters. A generalized assignment rule is introduced. A generalized procedure call rule is also introduced.

[18] In this paper a method is developed which is suitable for jumps and functions. The example used is the logarithmic search of an array of elements in sorted order. The result returned is the position in the array where the element
occurs.

[19] An ALGOL-like language is defined with conditional, while, procedure calls, and blocks. I/O, jumps, functions, and data structures are not included. Interpretive and axiomatic semantics are defined. This axioms system is proven sound and complete with respect to the given semantics. This paper gives a careful treatment of procedures using global variables.

[20] The equivalence of iterative and recursive definitions are proven. A general theorem on equivalence is also discussed and proven (general means that the theorem can be applied to more than one case).

[21] A two-way reduction among the intermittent assertions and the intermediate assertions methods is presented in this paper. It compares two induction principles called “always” and “sometime” for proving inevitability properties of programs.

[22] This paper described an invitational workshop called FM98.

[24] This paper serves the purpose of proving that Floyd’s inductive assertion method is complete. The proof is extended to programs with parameterless recursion. The relational approach is used throughout this paper in the definitions and the proofs. The paper applies its result to the towers of Hanoi problem.

[23] This is a book for which the aim was to bring the disciplines of program verification and program design together. A solid mathematical foundation is
provided for the technique of proving program correctness.

[25] This paper presents an argument that program verification is bound to fail since a social process cannot be developed among program verifiers.

[26] The intention of this book was to publish a set of algorithms in such a way that the reader would appreciate them.

[28] This paper extends the rule of inference for procedure calls to allow actual variable parameters to occur in actual value parameters, the body of a procedure to contain global variables that do not occur in assignment positions, and postconditions and "internal" assertions of a procedure to refer to the initial values of variable parameters.

[29] This paper basically gives arguments that program verification will never be completely successful.

[30] This paper provides a basis for defining the meanings of programs so that proofs of correctness, equivalence, and termination of programs can be constructed. This paper defines a flowchart language for expressing programs. A formal definition of an interpretation for flowcharts is defined and a verification of an interpretation of a flowchart is defined. Axioms are presented which define semantics. A flowchart language is defined for usage in the paper. The possibility of extending a partially specified interpretation into a completely specified interpretation is shown. A definition of well founded sets and a method
of proving termination of programs using well founded sets is described.

[31] This book is important because it explains program verification in such a way that a person just starting to learn program verification can grasp the different concepts and at the same time it covers a significant amount of material. It is intended as a textbook and has several exercises at the end of each chapter. The book tends to start simple then build up to more complex programming language features. The book starts by covering the simplest class of deterministic programs. The book includes a comprehensive coverage of some verification techniques for programs with procedures including directly recursive procedures. The verification of nondeterministic, concurrent, and distributed programs is also covered in this book.

[32] This paper defines an alternative approach to verification and compares it with Hoare's rules. Along with that a notion of correctness-preserving program transformations is introduced.

[33] The purpose of this paper is to point out that the new programming methodologies such as formal specification, systematic construction, and correctness proving are still fallible. Errors, inconsistencies, or confusing points are noted in a variety of published algorithms, many of which are being used as examples in formulating or teaching principles of such modern programming methodologies. Possible causes of these errors are pinpointed and general guidelines are given
which might help reduce the number of errors.

[34] This paper describes the prototype system called VISTA which provides assistance in synthesizing correct inductive assertions. The components of VISTA are described and their contribution to the whole system. VISTA is capable of generating a useful set of inductive assertions, given input and output specifications. VISTA was not fully implemented when this paper was written.

[35] The contribution of this paper is an extension to include multiple assignment to several subscripted variables. Arguments are given that the use of subscripted variables can lead to exponential explosion of the length of a proof.

[36] This paper accomplishes 2 things. First the multiple assignment statement is defined. The second accomplishment of this paper is a procedure call proof rule that is understood in terms of multiple assignment. Proof rules are developed for calls of procedures using global variables, var parameters, result parameters and value parameters. An attempt is made to clarify some issues that have arisen concerning the use of rules of inference to aid in generating “verification conditions” in mechanical verifiers and the use of logical variables to denote initial values of program variables.

[39] This paper presents a version of Hoare’s system for proving partial correctness of recursive programs. An extension is presented to the system which includes the rules their inverses and some other extensions.
[38] This paper surveys a number of known methods of proving programs within a simple set of deterministic programs. (key words: data-directed, syntax-directed, invariant assertion, intermittent assertions).

[40] This paper presents a logic formalism and a syntax for the formalism. It is claimed that first order logic is not powerful enough to express all schemas for use in program verification and therefore the formalism is second order.

[41] This paper explores the logical foundations of computer programming by using techniques first applied in the study of geometry. A set of axioms and inference rules of inference are described. These are the rule of consequence, rule of composition and the rule of iteration. The axioms and rules of inference in this paper assume no side effects with the evaluation of expressions. The purpose of the paper is to provide a logical basis for proofs of the properties of programs. The benefits for formal language definitions from the definition of axioms is discussed. It is also argued that axioms may provide a simple solution to leaving certain aspects of a language undefined.

[42] This paper deals in the verification of programs with procedures. It claims to reveal the ease of simultaneously proving program correctness and attaining high efficiency. This is provided that the programmer conforms to a natural discipline with respect to parameter passing.

[43] In this paper a proof is given of the correctness of an algorithm called
"Find." An informal description is given of the purpose of the program and the method used. A systematic technique is described for constructing a proof of a program during the process of writing it. The proof of termination is treated separately.

[44] This paper suggests a method of simplifying proofs of programs. The paper suggests an automatic method for accomplishing the transition between an abstract and a concrete program. Functions with side effects are covered.

[45] A proposed axiomatic definition is extended and applied to define the meaning of the programming language Pascal. Real arithmetic and goto statements are not covered. Data types like array types, file types, data types, pointer types are covered. Procedures with parameters are also covered. Flow diagrams are used to describe the language.

[46] This paper discuss automatic program verification. Defining the semantics of programming languages by axioms and rules of inference yields a deduction system within which proofs may be formed. A subgoaler for the deduction system is described whose input is a subset of Pascal programs plus inductive assertions. The output is a set of verification conditions. Several nontrivial arithmetic and sorting programs have been shown to satisfy specifications by using an interactive theorem prover to automatically generate proofs. Additional components for a more powerful verification system are under construction.
[47] In this paper general correctness is defined for both weakest precondition and strongest postconditions. The paper is a study of some simple theory with respect to predicate transformer semantics. It uses the relational model of programs.

[48] This paper concentrates on termination of programs and discusses several methods of proving termination. Discussed are Floyd approach, using the "no-infinitely-descending-chain" property of well founded sets, the loop approach the exit approach, and the data approach. The paper indicates the relative merit of each method.

[49] A method is suggested which involves conducting a logical analysis of programs by using invariants which express what is actually occurring in the program. The first part is devoted to techniques for automatic generation of invariants. The second part provides criteria for using the invariants for correctness and incorrectness simultaneously. The third part examines the implications of the approach for the automatic diagnosis and correction of logical errors.

[50] This paper shows the differences and the similarities of several methods of proving program correctness. A Relational semantic model for programming languages is introduced.

[51] Hoare's axiomatic method is applied in describing side effects and general jumps. Proof rules are introduced for variables, constants, unary operators and
binary operators. Proof rules for exit statements, label declarations, and labelled statements are also introduced. As for side effects a distinguished variable is introduced into the associated propositions which represents the value returned by an expression.

[52] This paper presents a method of verifying programs that manipulate data structures. Some key words are data structure, inductive assertion, marking algorithm, programming language, semantics.

[54] This paper deals with the construction of a program by constructing another program, proving it correct, then making a finite number of refinements with each refinement preserving correctness.

[55] This paper deals with the verification of programs with goto statements.

[56] This is a book on program verification and its foundations. Issues discussed are flowchart programming, while-programming, recursive programs, denotational semantics, inductive assertions, well founded sets, subgoal induction, structural induction, soundness and relative completeness.

[57] This paper describes the programming language of Euclid. Euclid is intended for the expression of system programs which are to be verified. All constructs are covered except storage allocation and machine dependencies. The proof rules for the verification of Euclid programs is also described.

[58] In this paper a method is presented for the verification of arrays, Records,
and pointers in Pascal. A proof rule has also been given for the Pascal storage allocation operation NEW.

[59] This paper describes an algorithm for converting a program $P$ into a first order formula $Wp$. In this way the problem of proving a program correct is reduced to proving the satisfiability of the first order formula. The equivalence of programs is also discussed providing a method for proving a program correct by proving its equivalence to another program that is known to be correct.

[62] In this paper an algorithm is presented for constructing formulas for functional programs defined by LISP like conditional recursive expressions. Such a formula produced, for a program $P$, will characterize the execution of the program. The problem of proving the correctness of a particular program with respect to a given assertion is reduced to proving the validity of the formula produced.

[60] This paper shows that it is possible to express most properties of algorithms in terms of partial correctness. The paper goes into theory for both deterministic and nondeterministic programs. It is claimed that the result achieved is of special interest since partial correctness has already been defined for many classes of deterministic programs.

[63] This paper presents, for a class of while programs, an axiomatic approach, similar to Hoare’s approach, which is designed to prove the total correctness of
algorithms. The difference lies in the possibility of proving both partial correctness and termination together. It is possible to prove total correctness in a single step or in several steps with each step proving more properties of the program.

[61] This paper discusses the many inductive methods for proving the correctness of programs. Examined are some known methods for proving properties of recursive programs. Some of the issues discussed are least fixed-point, computational induction and structural induction. It is claimed that two of these methods form a natural basis for future automatic program verifiers.

[64] This paper explores a technique, called intermittent assertions, for proving correctness and termination of programs simultaneously. It is shown that a proof of correctness by any of the conventional techniques can be rephrased as a proof by intermittent assertions. Finally it is shown how the intermittent assertion method can be applied to prove the validity of program transformations and the correctness of program transformations.

[65] In this paper an alternative approach is presented which makes explicit reference to the states of a computation. A distinction is made between an expression, its value and the values location.

[66] A proof rule for the procedure call is derived for procedures with value, result and value-result parameters. It is extended to procedures with unrestricted global variables and to recursive procedures. Like Gries's proof rule, it is based
on the substitution rule for assignment. However, it is more general and much simpler to apply.

[67] This paper presents a complete axiomatization of termination assertions for a rich class of programs. This includes recursive programs.

[70] This paper presents an inductive method for proving properties of recursive functions.

[71] A proof method called subgoal induction is presented in this paper. The relation between subgoal induction and other commonly used induction rules is discussed. A set of sufficient conditions are presented guarantee that a given input-output specification is strong enough for the use of subgoal induction.

[72] This paper discusses snapshots and shows how they can be used in program verification. General snapshots are also shown to be useful in constructing algorithms. It is shown that this method can be used to prove the correctness of an algorithm written in ALGOL 60.

[73] This paper is a critique of Hoare's logic. It defines a higher standard of correctness than conventionally used and judges the Hoare style logic by these standards. Some key words are soundness, partial correctness, proofs, defined functions, Goto, and Logic.

[74] This paper brings together the issues of expressiveness and rule of adaptation and connects them through an investigation of their soundness and
relative completeness.

[75] This paper appears to argue that London's certification of algorithm 245 does not take enough into consideration to be considered legitimate. Some key words are proof of algorithms, debugging, certification, meta-theory, sorting and in-place sorting.

[78] A notation for total correctness of programs with respect to input and output formulas is introduced. Hoare's loop axiom is rearranged in such a way as to form a good system for total correctness.

[79] In this paper an attempt is made to deal with mutual recursion and the termination of procedures with mutual recursion. A new proof rule is proposed to deal with recursion. This rule has a form of a recursion depth counter. Predicate transformers are also discussed. The validity and completeness of the rule is established.

[80] This paper introduces a decomposition rule for Hoare's logic.

[81] This paper attempts to handle partial and total correctness within a unified framework. Covered are goto statements and recursive procedures.

[82] This paper discusses methods for mechanically synthesizing loop predicates. Two classes of techniques are considered: (1) Heuristic methods which derive loop predicates from boundary conditions and/or partially specified inductive assertions: (2) extraction methods which use input predicates and appropriate
weak interpretations to obtain certain classes of loop predicates by an evaluation on the weak interpretation.

[83] This paper describes a language called Alphard which is designed to support both structured programming and program verification. (Keywords: abstract data types, abstraction and representation, assertions, correctness, information hiding, levels of abstraction, modular decomposition, program verifications)
Bibliography


VITA AUCTORIS

Matthew Blain was born in 1967 in Windsor, Ontario. He graduated from Brennan High School in 1987. From there he went on to the University of Windsor where he obtained a B. C. S. (Bachelor of Computer Science) in 1992. He is currently a candidate for the Master's degree in Computer Science at the University of Windsor and hopes to graduate in spring of 1995.