The South African Institute of Computer Science
and
Information Technology

The 1997 National
Research and
Development
Conference

Riverside Sun
Vanderbijlpark
13 & 14 November

Hosted by

Potchefstroomse Universiteit
vir Christelike Hoër Onderwys

The Department of Computer Science and Information Systems
Potchefstroom University for Christian Higher Education
Vaal Triangle Campus

PROCEEDINGS

Edited by L.M. Venter & R.R. Lombard
Proceedings
of the
The 1997 National
Research and
Development
Conference
Towards 2000

Riverside Sun
Vanderbijlpark
13 & 14 November

Edited by
L.M. Venter
R.R. Lombard
Foreword

This book contains a collection of papers presented at a Research and Development conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT). The conference was held on 13 & 14 November 1997 at the Riverside Sun, Vanderbijlpark. Most of the organization for the conference was done by the Department of Computer Science and Information Technology of the Vaal Triangle Campus, Potchefstroom University for Christian Higher Education.

The programming committee accepted a wide selection of papers for the conference. The papers range from detailed technical research work to reports of work in progress. The papers originate mainly from Academia, but also describe work done in and for Industry. It is hoped that the papers give a true reflection of the current research scene in Computer Science and Information Technology in South Africa. Since one of the aims of the conference is Research development, the papers were not subjected to a refereeing process.

A number of people spent numerous hours helping with the organization of this conference. In this regard, we wish to thank the members of the Organizing committee, and the Programming committee who had very little time to screen the abstracts and compile the program. A special thanks goes to the secretary of the department, Mrs Helei Jooste, whose very able work was interrupted by the birth of her first child.
Organizing Committee

Conference General Chairs
Prof. J.M. Hattingh (PU for CHE)

Organizing Chair
Prof. Lucas Venter (PU for CHE)

Organizing Committee
Mrs. S. Gilliland
Mr. J.P. Jooste
Mr. R.R. Lombard
Mrs. M. Huisman

Secretariat
Mrs. H. Jooste

Program Chair
Prof A de Waal (PU for CHE)

Program Committee
Prof D. Kourie (UP)
Prof. C. Bornman (UNISA)
Prof. L.M. Venter (PU for CHE)
## Table of Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>i</td>
</tr>
<tr>
<td>Organizing Committee</td>
<td>ii</td>
</tr>
<tr>
<td>List of Contributors</td>
<td>vii</td>
</tr>
<tr>
<td>Software Objects Change: Problems and Solution</td>
<td>1</td>
</tr>
<tr>
<td>S.A. Ajila</td>
<td></td>
</tr>
<tr>
<td>Liming-like Curve Constructions</td>
<td>26</td>
</tr>
<tr>
<td>M.L. Baart and R. McLeod</td>
<td></td>
</tr>
<tr>
<td>A Model for Evaluating Information Security</td>
<td>27</td>
</tr>
<tr>
<td>L. Barnard and R. von Solms</td>
<td></td>
</tr>
<tr>
<td>Integrating Spatial Data Management and Object Store Technology</td>
<td>31</td>
</tr>
<tr>
<td>S. Berman, S. Buffler and E. Voges</td>
<td></td>
</tr>
<tr>
<td>Metamodelling in Automated Software Engineering</td>
<td>32</td>
</tr>
<tr>
<td>S. Berman and R. Figueira</td>
<td></td>
</tr>
<tr>
<td>Using Multimedia Technology for Social Upliftment in Deprived</td>
<td>33</td>
</tr>
<tr>
<td>Communities of Southern Africa</td>
<td></td>
</tr>
<tr>
<td>L. Bester and E. de Preez</td>
<td></td>
</tr>
<tr>
<td>Extending the Client-Server Model for Web-based Execution of</td>
<td>36</td>
</tr>
<tr>
<td>Applications</td>
<td></td>
</tr>
<tr>
<td>L. Botha, J.M. Bishop and N.B. Serbedzija</td>
<td></td>
</tr>
<tr>
<td>Access Control Needs in an Electronic Workflow Environment</td>
<td>45</td>
</tr>
<tr>
<td>R.A. Botha</td>
<td></td>
</tr>
<tr>
<td>The Use of the Internet in an Academic Environment to Commercially</td>
<td>51</td>
</tr>
<tr>
<td>Supply and Support Software Products</td>
<td></td>
</tr>
<tr>
<td>B. Braude and A.J. Walker</td>
<td></td>
</tr>
<tr>
<td>Explanation Facilities in Expert Systems Using Hypertext Technology</td>
<td>63</td>
</tr>
<tr>
<td>T. Breetzke and T. Thomas</td>
<td></td>
</tr>
<tr>
<td>Theoretical Computer Science: What is it all about, and is it of any</td>
<td>75</td>
</tr>
<tr>
<td>relevance to us?</td>
<td></td>
</tr>
<tr>
<td>C. Brink</td>
<td></td>
</tr>
<tr>
<td>Representing Quadrics on a Computer</td>
<td>76</td>
</tr>
<tr>
<td>M.A. Coetzee and M.L. Baart</td>
<td></td>
</tr>
</tbody>
</table>
The Generation of Pre-Interpretations for Detecting Unsolvable Planning Problems
D.A. de Waal, M. Denecker, M. Bruynooghe and M. Thielscher

The Emerging Role of the Chief Information Officer in South Africa
B. Dekenah

A Java-Implemented Remote Respiratory Disease Diagnosis System on a
High Bandwidth Network
A. Foster

Early Results of a Comparative Evaluation of ISO 9001 and ISO/IEC
15504 Assessment Methods Applied to a Software Project
C. Gee and A.J. Walker

A Neural Network Model of a Fluidised Bed
M. Hajek

The Effects of Virtual Banking on the South African Banking Industry
M.L. Hart and M. Dunley-Owen

Linear Response Surface Analysis and Some Applications
J.M. Hattingh

Model Checking Software with Symbolic Trajectory Evaluation
A. Hazelhurst

A Risk Model to Allocate Resources to Different Computerized Systems
H.A. Kruger and J.M. Hattingh

Returns on the Stock Exchange
J.W. Kruger

Cardinality Constrained 0-1 Knapsack Problems
M.F. Kruger, J.M. Hattingh and T. Steyn

An Investigation in Software Process Improvement in the
Software Development of a large Electricity Utility
M. Lang and A.J. Walker

Design and Implementation of a C++ Package for Two-Dimensional
Numerical Integration
D.P. Laurie, L Pluym and Ronald Cools

Algebraic Factorization of Integers Using BDE's
H. Messerschmidt and J. Robertson
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Optimization of Routes after the Process of Recovery</td>
<td>176</td>
</tr>
<tr>
<td>M. Mphahlele and J. Roos</td>
<td></td>
</tr>
<tr>
<td>Using a Lattice to Enhance Adaptation Guided Retrieval in Example</td>
<td>177</td>
</tr>
<tr>
<td>Based Machine Translation</td>
<td></td>
</tr>
<tr>
<td>G.D. Oosthuizen and S.L. Serutla</td>
<td></td>
</tr>
<tr>
<td>Information Systems Development and Multi Criteria Decision</td>
<td>192</td>
</tr>
<tr>
<td>Making / Systems Thinking</td>
<td></td>
</tr>
<tr>
<td>D. Petkov, O. Petkova</td>
<td></td>
</tr>
<tr>
<td>The Development of a Tutoring System to Assist Students to Develop</td>
<td>193</td>
</tr>
<tr>
<td>Answering Techniques</td>
<td></td>
</tr>
<tr>
<td>N Pillay</td>
<td></td>
</tr>
<tr>
<td>Combining Rule-Based Artificial Intelligence with Geographic</td>
<td>194</td>
</tr>
<tr>
<td>Information Systems to Plan the Physical Layer of Wireless Networks in</td>
<td></td>
</tr>
<tr>
<td>Greenfield Areas</td>
<td></td>
</tr>
<tr>
<td>K. Prag, P. Premjeeth and K. Sandrasegaran</td>
<td></td>
</tr>
<tr>
<td>A Distributed Approach to the Scheduling Problem</td>
<td>202</td>
</tr>
<tr>
<td>V. Ram and P. Warren</td>
<td></td>
</tr>
<tr>
<td>More readings than I thought: Quantifier Interaction in Analysing the</td>
<td>203</td>
</tr>
<tr>
<td>Temporal Structure of Repeated Eventualities</td>
<td></td>
</tr>
<tr>
<td>S. Rock</td>
<td></td>
</tr>
<tr>
<td>Ray Guarding Configuration of Adjacent Rectangles</td>
<td>221</td>
</tr>
<tr>
<td>I. Sanders, D. Lubinsky and M. Sears</td>
<td></td>
</tr>
<tr>
<td>Developing Soft Skills in Computer Students</td>
<td>239</td>
</tr>
<tr>
<td>C Schröder, T. Thomas</td>
<td></td>
</tr>
<tr>
<td>Information Security Awareness, a Must for Every Organization</td>
<td>250</td>
</tr>
<tr>
<td>M. Thomson and R. von Solms</td>
<td></td>
</tr>
<tr>
<td>Pla Va: A Lightweight Persistent Java Virtual Machine</td>
<td>253</td>
</tr>
<tr>
<td>S Tjasink and S. Berman</td>
<td></td>
</tr>
<tr>
<td>Beliefs on Resource-Bounded Agent</td>
<td>267</td>
</tr>
<tr>
<td>E. Viljoen</td>
<td></td>
</tr>
<tr>
<td>Object-Orientated Business Modelling and Re-engineering</td>
<td>268</td>
</tr>
<tr>
<td>M. Watzenboeck</td>
<td></td>
</tr>
</tbody>
</table>
On Indexing in Case Based Reasoning Applied to Pre-Transportation Decision Making for Hazardous Waste Handling
K.L. Wortmann, D. Petkov and E. Senior

Author Index
List of Contributors

S.A. Ajila
Department of Mathematics and Computer Science
National University of Lesotho
Roma, 180
Lesotho

L. Baart
Department of Mathematics
Vaal Triangle Campus of the PU for CHE
PO Box 1174
Vanderbijlpark, 1900

L. Barnard
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

S. Berman
University of Cape Town
Rondebosch, 7701

L. Bester
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

J.M. Bishop
Computer Science Department
University of Pretoria
Pretoria, 0002

L. Botha
Computer Science Department
University of Pretoria
Pretoria, 0002

R.A. Botha
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

B. Braude
Software Engineering Applications Laboratory,
Electrical Engineering
University of the Witwatersrand
Private Bag 3
Wits, 2050

T. Breetzke
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

C. Brink
University of Cape Town
Rondebosch, 7700

M. Bruynooghe
Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee
Belgium

S. Buffler
University of Cape Town
Rondebosch, 7701

M.A. Coetzee
Department of Mathematics
PU for CHE
Private Bag X6001
Pochefstroom, 2520

R. Cools
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee
Belgium

E. de Preez
Faculty of Computer Studies
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth, 6000

D.A. De Waal
Department of Computer Science and Information Systems
PU for CHE
Private Bag X6001
Pochefstroom, 2531

B. Dekena
The Board of Executors

M. Denecker
Departement Computerwetenschappen
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee
Belgium

M. Dunley-Owen
Department of Information Systems
University of Cape Town
Rondebosch, 7700

R. Fiqueira
University of Cape Town
Rondebosch, 7701

A. Foster
Department of Computer Science
University of Cape Town
Rondebosch, 7701

C. Gee
Software Engineering Applications Laboratory,
Electrical Engineering
University of the Witwatersrand
Private Bag 3
Wits 2050

vii
M. Hajek  
Department of Computer Science  
University of Durban Westville  
Private Bag X54001  
Durban, 4000

M.L. Hart  
Department of Information Systems  
University of Cape Town  
Rondebosch, 7700

J.M. Hattingh  
Department of Computer Science and Information Systems  
PU for CHE  
Private Bag X6001  
Potchefstroom, 2520

S. Hazelnurst  
Department of Computer Science  
University of the Witwatersrand  
Private Bag 3  
Wits 2050

H.A. Kruger  
Department of Computer Science and Information Systems  
PU for CHE  
Private Bag X9001  
Potchefstroom, 2520

J.W. Kruger  
University of the Witwatersrand  
Private Bag 3  
Wits, 2050

M.F. Kruger  
PU for CHE  
Private Bag X6001  
Potchefstroom, 2520

M.T. Lang  
Eskom Information Technology Department

D. Laurie  
Department of Mathematics  
Vaal Triangle Campus of the PU for CHE  
PO Box 1174  
Vanderbijlpark, 1900

D. Lubinsky  
Department of Computer Science  
University of the Witwatersrand  
Private Bag 3  
Wits, 2050

R. McLeod  
Saltire Software Inc.  
Tigard  
Oregon  
U.S.A

H.J. Messerschidt  
Department of Computer Science and Informatics  
University of the Orange Free State  
PO Box 339  
Bloemfontein, 9300

M. Mphahlele  
Department of Computer Science  
University of the North  
Private Bag X1106  
Sovenga, 0727

G.D. Oosthuizen  
Department of Computer Science  
University of Pretoria  
Pretoria, 0002

J. Owen  
University of Cape Town  
Rondebosch, 7701

D. Petkov  
Department of Computer Science  
University of Natal  
Private Bag X01  
Scotsville, 3209

O. Petkova  
Technikon Natal  
PO Box 101112  
Scotsville, 3209

N. Pillay  
Department of Financial Studies  
Technikon Natal, Pietermaritzburg  
PO Box 101112  
Scotsville, 3209

L. Pluym  
Katholieke Universiteit Leuven  
Celestijnenlaan 200A  
B-3001 Heverlee  
Belgium

K. Prag  
Department of Electrical Engineering  
University of Durban-Westville  
Private Bag X54001  
Durban, 4000

P. Premjeeth  
Department of Electrical Engineering  
University of Durban-Westville  
Private Bag X54001  
Durban, 4000

V. Ram  
Department of Computer Science  
University of Natal  
Private Bag X01  
Scotsville, 3209

J. Robertson  
Department of Computer Science and Informatics  
University of the Orange Free State  
PO Box 339  
Bloemfontein, 9300

S. Rock  
Department of Artificial Intelligence  
Edinburgh University  
United Kingdom

J. Roos  
Department of Computer Science  
University of Pretoria  
Pretoria, 0002

I. Sanders  
Department of Computer Science  
University of the Witwatersrand  
Private Bag 3  
Wits, 2050
K. Sandrasegaran  
Department of Electrical Engineering  
University of Durban-Westville  
Private Bag X54001  
Durban, 4000

C. Schoder  
Faculty of Computer Studies  
Port Elizabeth Technikon  
Private Bag X6011  
Port Elizabeth, 6000

M. Sears  
Department of Mathematics  
University of the Witwatersrand  
Private Bag 3  
Wits, 2050

E. Senior  
International Center for Waste Technology  
University of Natal, Pietermaritzburg  
Private Bag X01  
Scotsville, 3209

N.B. Serbedzija  
GMD FIRST  
Rudower Chaussee 5  
D-12489 Berlin  
Germany

S.L. Serutla  
Department of Computer Science  
The University of Pretoria  
Pretoria, 0002

T. Steyn  
PU for CHE  
Private Bag X6001  
Potchefstroom, 2520

M. Thielscher  
Fachgebiet Intellektik, Fachgebiet Informatik  
Technische Hochschule Darmstadt  
Alexanderstrasse 10  
D-64283 Darmstadt  
Germany

T. Thomas  
Faculty of Computer Studies  
Port Elizabeth Technikon  
Private Bag X6011  
Port Elizabeth, 6000

M. Thom-  
Faculty of Computer Studies  
Port Elizabeth Technikon  
Private Bag X6011  
Port Elizabeth, 6000

S. Tjasink  
University of Cape Town  
Rondebosch, 7700

E. Viljoen  
Department of Computer Science and  
Information Systems  
University of South Africa  
PO Box 392  
Pretoria, 0001

E. Voges  
University of Cape Town  
Rondebosch, 7701
Model Checking Software with Symbolic Trajectory Evaluation
Extended Abstract

Scott Hazelhurst
Programme for Highly Dependable Systems
Department of Computer Science
University of the Witwatersrand, Johannesburg,
2050 Wits, South Africa
scott@cs.wits.ac.za

October 6, 1997

Abstract

Traditional methods of testing computer systems, although valuable, are inadequate for ensuring sufficiently high quality in systems in which a high degree of reliability must be placed. Formal methods for development of hardware and software offer considerable promise in improving quality. Symbolic methods for model checking (e.g. [13]) have had considerable success in the verification of hardware, and a number of industrial concerns use such methods in their design process.

The verification of software is much more difficult. Automatic model checking is attractive because it is automatic, but the computational cost may be prohibitive, motivating research with the ultimate goal of finding (relatively) automated, tractable methods of verifying software.

Symbolic trajectory evaluation (STE) may be a good starting point since it is able to deal with very large state spaces. This paper presents the first step in the research project of extending the theory of STE to a framework suitable for verifying software.

1 Introduction

Traditional methods of testing computer systems, although valuable, are inadequate for ensuring sufficiently high quality of software for systems which require high reliability. Formal methods for development of hardware and software offer considerable promise in improving quality, and a much higher degree of confidence can be placed in systems that have been formally verified. However, verification is a very difficult computational problem, often resulting in methodologies that are difficult to use and which require considerable computational resources.

Significant progress has been made with the verification of hardware. Symbolic methods for model checking (e.g. [3, 13]) have had considerable success in the verification of hardware, and a number of industrial concerns use such methods in their design process. Automatic model checking algorithms are attractive because a large part of the verification process is automated, which makes them suitable for designers to use without extensive training in formal methods. See [11] for some examples of hardware verification methodologies.

The verification of software is much more difficult. Methods of verifying software based on Hoare-style reasoning are very powerful, but may require considerable human intervention. Automatic model checking is attractive because it is automatic, but it is important to find methods that require reasonable computational resources.

In general, fully automated verification methodologies are not possible, and even when significant restrictions are made on programs it does not seem likely that fully automated methods for software verification will appear soon. However, the great advances in theory and practice of verification over the last decade indicate that practical verification methodologies are possible.
Goal of this paper

Symbolic trajectory evaluation (STE) is a verification methodology that has been successfully applied to hardware designs [13]. STE has shown that its strength is representing systems with very large state spaces. Together with its associate compositional theory, STE can be used to verify designs of some complexity and size with a reasonable mix of automated steps and human intervention. For this reason, an investigation of how STE and the ideas learned in applying STE to hardware verification can be applied to software verification is worthwhile.

The purpose of this paper is to lay the theoretical basis for using STE to verify software. The basic engine of STE is a symbolic simulation algorithm coupled with a novel scheme for data abstraction which can use good data structures for efficient implementation. There have been previous studies using symbolic simulation for verification (eg, [14]) and although some success has been achieved, the computational demands of the problem are extremely daunting, which motivates an exploration of whether new methods of representing state will be effective.

A key change is that in hardware, a temporal view of the behaviour of the system is useful; in software this shifts to a spatial view, where we are interested in the state of the program at various control points. This requires two significant changes to STE theory:

- Assertions based on antecedent/consequent pairs rather than a temporal logic become the appropriate specification language; and
- The model checking algorithm must be generalised to deal with the fact that there is not, in general, a linear ordering of control points in a program.

The work presented in this paper is work in progress and so is incomplete and unpolished in parts.

Outline of paper

Section 2 describes how programs are represented – how the models are constructed. Modified finite state machines are used for representation and are built from programs written in a simple general imperative programming language. Section 3 presents the language used for specification: assertions based on a program logic. Section 4 describes a verification method based on symbolic trajectory evaluation. Section 5 then extends these results to allow symbolic representation of assertions, thereby giving a much richer specification language. Section 6 discusses how the ideas of this paper can be implemented in a practical verification system. The research presented in this paper is research in progress and Section 7 concludes by indicating how the next steps in the research.

2 Program Graphs

This section discusses how programs are represented. Section 2.1 presents the simple programming language in which programs may be written. Sections 2.2 to 2.4 then describe how the program is represented formally—effectively an extended finite state machine. Section 2.2 starts by showing how the static structure of the program is formally represented. Section 2.3 discusses data representation, which is key for efficient verification. Then, the formal representation of the dynamic behaviour of a program is described in Section 2.4. Finally, Sections 2.5 and 2.6 establish some mathematical machinery which will be useful at a later stage.

2.1 Syntax and semantics of programming language

The exact details of the programming language is not given because it is not particularly important. The language is a Pascal-like language that supports:

- assignment of expression values to variables
- an if-then-else construct
• a while loop

A standard semantics for all of these is assumed, as is that standard Floyd-Hoare proof rules hold. The following restrictions are made:

• The language does not contain procedures.

• All loops must come with putative loop invariants and must be guaranteed to terminate. That the given loop invariants are invariants will be checked.

Neither of these restrictions is desirable. The lack of procedures is a particular problem. However, the purpose of this paper is to lay down the basis for STE-based verification and so this paper is only a starting point. The question of how procedures can be dealt with will be discussed in Section 6.

2.2 Constructing a program graph

The static structure of a program is represented by a program graph constructed from the program text. This section describes how a program graph can be constructed from a program. Let statements \( S, S_1 \) and \( S_2 \) be represented by program graphs \( (V, E), (V_1, E_1), (V_2, E_2) \). By the construction that follows any program graph \( (V, E) \) has a unique source vertex (denoted \( \text{src}(V, E) \)) that has no edges entering it, and a unique end vertex (denoted \( \text{end}(V, E) \)) that has no edges leaving it. Each vertex can be uniquely identified. The vertices of the graph are called control points because they can be used to describe the flow of control of the program. The edges show how control flows, and are labelled with conditions which indicate when control may move from one control point to another.

The following inductively defined rules can be used to construct a program graph from an arbitrary program segment. At the same time as constructing the graph we also construct a partial order, \( \leq \), that reflects dependencies of control points. If the path order relates two points \( x \) and \( y \) \( (x \leq y) \), then an execution of the program that includes \( x \) will include \( y \) at a later stage.

1. The statement \( w := E \) is represented by the vertex \( v \), which is labelled by the action \( (w := E) \). We let \( v \leq v \).

2. For the statement if \( b \) then \( S_1 \) else \( S_2 \) we introduce two dummy nodes \( v_1 \) and \( v_2 \) (with null actions associated). \( v_1 \) is 'joined' to the source vertices of the program graphs of \( S_1 \) and \( S_2 \), with the edges being labelled with \( b \) and \( \sim b \) respectively to indicate when the edges should be taken. Similarly, the end vertices of the process graph are connected to \( v_2 \). The vertices \( v_1 \) and \( v_2 \) become the source and end vertices of the new graph.

Formally, the program graph of the statement is \( (V', E') \), where

\[
V' = \{v_1, v_2\} \cup V_1 \cup V_2
\]

\[
E' = \{(v_1, \text{src}(V_1, E_1), b), (v_1, \text{src}(V_2, E_2), \sim b), (\text{end}(V_1, E_1), v_2, t), (\text{end}(V_2, E_2), v_2, t)\}
\]

For the path order, we let \( x \leq y \) if either \( x \leq_1 y \) or \( x \leq_2 y \) or if \( x = v_1 \) and \( x = v_2 \).

3. For the statement while \( b \) do \( S \), we introduce four new dummy nodes \( v_1, \ldots, v_4 \) all labelled with null actions. \( v_1 \) and \( v_4 \) are the source and end vertices of the graph. An edge connects \( v_1 \) with \( v_2 \). An edge connects \( v_2 \) with the source vertex of \( S_1 \)'s program graph; this is labelled with \( b \) to indicate that this path should be taken when the loop's condition is true. An edge (labelled with \( t \)) connects the end vertex of \( S_1 \)'s program graph with \( v_3 \). An edge (labelled with \( t \)) connects \( v_3 \) to \( v_2 \) to return to the beginning of the loop. Finally an edge, labelled with \( \sim b \) connects \( v_2 \) with \( v_4 \) to indicate when looping must finish.

Informally these new dummy vertices represent:
v₁ The program about to enter the loop for the first time.
v₂ The program about to enter an iteration of the loop.
v₃ The program has just completed an iteration of the loop.
v₄ The program has just completed the loop.

Formally, the program graph of the statement is \((V', E')\), where
\[
\begin{align*}
V' &= \{v₁, \ldots, v₄\} \cup V \\
E' &= \{(v₁, v₂, t), (v₂, \text{src}(V, E), b), \text{(end}(V, E), v₃, t), (v₃, v₂, t), (v₂, v₄, \neg b)\} \cup E.
\end{align*}
\]

For the path order, we let \(x \preceq y\) if either \(x \preceq₁ y\) or \(x = v₂ \land y \in V \cup \{v₂, v₃\}\) or \(x \in V \cup \{v₂\} \land y = v₃\) or \(x = v₁ \land y = v₄\).

4. The program graph for the sequence \(S₁; S₂\) is constructed by connecting the end vertex of \(S₁\) with the source vertex of \(S₂\). The source vertex of \(S₁\) is the source vertex of the combined graph, and the end vertex of \(S₂\) is the end vertex of the new graph. Formally, the process graph is \((V', E')\), where:
\[
\begin{align*}
V' &= V₁ \cup V₂; \\
E' &= \{\text{(end}(V₁, E₁), \text{src}(V₂, E₂), t)\} \cup E₁ \cup E₂.
\end{align*}
\]

For the path order, we let \(x \preceq y\) if either \(x \preceq₁ y\) or \(x \preceq₂ y\) or \(x \in V₁ \land y \in V₂\).

As can be seen by the construction, each control point, \(vᵢ\), has an operation, \(\text{op} \ vᵢ\), associated with it. This operation is either an assignment or a null operation. The semantics of a program is given by the corresponding program graph, the semantics of which is described in Section 2.4.

Besides the control points determined by the above algorithm we add two special control vertices \(\text{err}\), representing the system entering an error state (usually caused by ill-defined interaction with the outside world) and \(\text{undef}\), the undefined control point, representing the system being at any one of the control points.

A note on the path order \(\preceq\):

It is important to note that it is not the case that if there is a path in the graph between \(u\) and \(v\) that \(u\) and \(v\) will necessarily be related by the path order. For example, in point 3 above \(v₁ \preceq v₄\), but it is not the case that \(v₁ \preceq v₂\). The path order also captures a notion of different levels of hierarchical control abstraction so the control points of the body of a loop are related to each other by the path order (according to what the body of the loop looks like), but they are not related to control points outside of the loop of the body.

A set of vertices \(V' \subseteq V\) is sensible if \(\preceq\) is a total order on \(V'\).

2.3 Values and partial orders

Representation of data is a critical issue for practical verification. All the variables have type. From the programmer's perspective, a type is a set of values, and the program may assign values of the appropriate type to a variable or modify an existing value.

When reasoning about a program, however, it turns out to be convenient to consider a variable as coming from some richer structure than a set of values. Such a richer structure is called a lattice. An example will help illustrate the concept, and then the principle will be formally defined.

Suppose that we have an integer variable \(x\). This allows the program to assign integer values in some finite range to \(x\) (in this example, assume that the integers are the set \(T \triangleq \{-15, \ldots, 0, \ldots, 15\}\) and that the program can set \(x\) to any value in \(T\). Two special values are introduced: \(U\) (pronounced bottom) and \(Z\) (pronounced top). The value \(U\) represents an unknown, uninitialised or 'don't care' value, thus, if \(x\) has the bottom value it is as if it had some arbitrary but unknown value. The top value \(Z\)
represents contradictory information. U and Z can be thought of as special integer values that can be used to represent our knowledge of the state of a variable.

Using these two values, we now construct the lattice \text{fluff}(T) from \( T \) by adding the values U and Z and defining the partial order \( U \sqsubseteq t, t \sqsubseteq t, t \sqsubseteq Z \) where \( t \) can be any value of \( T \) or U or Z. The resulting structure is a complete lattice under the partial order \( \sqsubseteq \). This is shown graphically in Figure 1.

![Lattice structure](image)

**Figure 1:** Lattice structure containing the values \(-15, -14, \ldots, 0, \ldots, 15\)

The partial order on this lattice is shown by the lines. The ordering is an information ordering or abstraction relation. The lower down in the lattice, the less information we have and the more abstract the view of the variables state; the higher up in the lattice, the more information we have. Note that the numbers themselves are not ordered with respect to this relation; all elements are related to the top and bottom elements only.

If \( x \) is the sole variable in the program, then the state space of the program is \( V \times \text{fluff}(T) \), where \( V \) is the set of control points in the program graph.

This idea can be generalised to any type \( T \); we add top and bottom elements to get \( \text{fluff}(T) \) with its partial ordering.

What value does an expression of type \( T \) have if it contains a variable with a top or bottom value? More generally, if we have an operation \( \odot \) on \( T \), how should its definition be extended to \( \text{fluff}(T) \)? Different decisions can be made for different types, depending on the meaning of the type. A good decision might be to allow any expression containing a Z to have a Z value, and otherwise to allow any expression containing a U value to have the value U. Take a simple example. What if we have

\[
x := y - z
\]

where the current value of \( y \) and \( z \) is U? Since U stands for any arbitrary number in the \( T \), \( U - U \) could be any arbitrary number in \( T \), which would seem to justify the suggestion above. Different choices could be made (and we shall see an important exception later).

Some mathematical background: A partial order over a set \( T \) is a relation (e.g. a subset of \( T \times T \)) that is reflexive, anti-symmetric and transitive. A lattice is pair \( (T, \sqsubseteq) \) where \( \sqsubseteq \) is a partial order over \( T \) and every finite subset of elements of \( T \) have a least upper bound (join) and a greatest lower bound (meet). A complete lattice is a lattice in which every subset of elements has a least upper bound and greatest lower bound.

It is worth mentioning that richer lattice structures are possible for representing data. This is a topic worth further investigation but is not in the scope of this paper.

Extending the partial order: Suppose \( x \) and \( y \) are of types \( T_1 \) and \( T_2 \) respectively. \( T_1 \) and \( T_2 \) can be extended to \( \text{fluff}(T_1) \) (with partial order \( \sqsubseteq_1 \)) and \( \text{fluff}(T_2) \) (with partial order \( \sqsubseteq_2 \)); they both have their own top and bottom elements. \( \text{fluff}(T_1) \times \text{fluff}(T_2) \) is also a complete lattice under the natural partial ordering \( (u_1, v_1) \sqsubseteq (u_2, v_2) \) exactly when \( u_1 \sqsubseteq u_2 \) and \( v_1 \sqsubseteq v_2 \). The top element of the new lattice is
(Z_1, Z_2), and the bottom element of the new lattice is (U_1, U_2). In this case, if V is the set of control points then the state space of the program is \( V \times (T_1 \times T_2) \).

This idea can be generalised so that if the variables of a program are \( x_1, \ldots, x_n \) of types \( T_1, \ldots, T_n \) then \( S = \text{fluff}(T_1) \times \ldots \times \text{fluff}(T_n) \) is a complete lattice under the natural partial ordering, and the state space of the program is \( V \times S \). Being a complete lattice, the join (least upper bound) of any two elements is well defined — the join of \( x \) and \( y \) is written \( x \cup y \).

The truth domain \( Q \): The truth domain will be discussed in detail later. Two truth values that can be used are \( t \) (truth) and \( f \) (falsity).

2.4 Evaluating a program graph

Suppose a program has the program graph \((V, E)\) and variables \( x_1, \ldots, x_n \) of types \( T_1, \ldots, T_n \). Thus, at any instant in time, the state of the variables is an element of \( \text{fluff}(T_1) \times \ldots \times \text{fluff}(T_n) \). For convenience, we let \( S \) denote \( \text{fluff}(T_1) \times \ldots \times \text{fluff}(T_n) \), \( U \) denote \((U_1, \ldots, U_n)\), which is the bottom element of \( S \), and \( Z \) denote \( S \)'s top element.

A program graph represents the static structure of the program. We also need to represent the dynamic behaviour of the program. This is represented by the next state function \( Y : V \times S \rightarrow V \times S \). This shows how in a given state (i.e. at particular control point, and with particular variable values) the program moves to a different control point with possibly different variable values.

\( Y \) is defined as follows. Suppose there is an edge \((v, w, b)\) such that \( b(s) = t\): then \( Y(v, s) = (w, s') \) where \( s' = (\text{op} v)(s) \). Note, that there will be at most one such edge so provided that such an edge exists, this is well defined. If no such edge exists, then \( Y \) is not defined. (Or go into error or undefined states)

2.5 Sequences, trajectories and abstraction

We shall need to manipulate sequences, and therefore adopt the following notation. If \( \sigma \) is the sequence \( t_0 t_1 t_2 \ldots t_k \), then \( \sigma_i = t_i, \sigma_{i+1} = t_i t_{i+1} \ldots t_k \) and \( \sigma_{j \geq i} = t_i t_{i+1} \ldots t_j \). The empty sequence is denoted \( \epsilon \). (Of course, sequences can be infinite. For convenience, we may abuse notation and write a sequence such as \( t_0 t_1 t_2 \ldots t_\infty \). This should be understood to represent an infinite sequence, and the reader should understand there is no 'final' element.)

We shall be considering sequences of states (each element of the sequence represents a control point and state of variables). The most interesting sequences are those that are related to the next state function of the program graph.

Definition 2.1. A trajectory is a sequence of states \((v_0, s_0)(v_1, s_1)\ldots(v_k, s_k)\) such that for \( i = 0, \ldots, k - 1 \), \( Y(v_i, s_i) = (v_{i+1}, s_{i+1}) \) for some \( t_{i+1} \subseteq s_{i+1} \).

Trajectories are therefore possible executions of the program. This notion can be generalised.

Definition 2.2. A run is a sequence of states \((v_0, s_0)(v_1, s_1)\ldots(v_k, s_k)\) such that for \( i = 0, \ldots, k - 1 \), there is a trajectory \((v_i, s_i)\ldots(v_{i+1}, s_{i+1})\).

In other words it must be possible to go from one state in a run to the next state in a run by executing the next state function (possibly many times). In a sense a run is an abstraction of a trajectory in that it is a possible run of the program, but some of the details of how the run was obtained are missing.

We can generalise the notion of abstraction in the following way.

Definition 2.3. Let \( \sigma^1 = (v_0, s_0)(v_1, s_1)\ldots \) and \( \sigma^2 = (w_0, t_0)(w_1, t_1)\ldots \) be sequences. \( \sigma^1 \) is an abstraction of \( \sigma^2 \) (written \( \sigma^1 \sqsubseteq \sigma^2 \)) if:

1. \( \forall i, \exists j_i \ni v_i = w_{j_i} \) and \( s_i \sqsubseteq t_{j_i} \), with \( j_i \leq j_{i+1} \).
2. \( \forall i, k = j_{i-1} + 1, \ldots, j_i - 1, v_j \neq w_i \)
The first condition says the things that happen in $u^1$ also happen in $u^2$. The second condition ensures that there are no inconsistencies by insisting that control points that appear in $u^1$ appear in $u^2$ in a way consistent with the way they appear in $u^1$. If $u^1 \subseteq u^2$ and $u^2$ is a run, then so will $u^1$ be a run: it captures computational information of $u^2$ but misses out some of the details of how that computation was done.

2.6 Manipulating states and sets of states

We shall see later that a set of states is a useful way of representing some properties. Let $A \subseteq V \times \text{fluff}(T)$ be a set of states.

$A$ is sensible if $\leq$ is a total order on $\{v : (v, s) \in A\}$. If $A$ is sensible then in any trajectory of the program, the control points in $A$ will, if executed, be executed in the order specified in $A$.

Let $\sigma^1$ and $\sigma^2$ be sequences of states for the program graph $(V, E)$. If the control points in $\sigma^1$ and $\sigma^2$ are collectively totally ordered by the path order, we define $\sigma^1 \cup \sigma^2$ to denote the combination of the information in both sequences. The path order indicates in which order the control points must appear and so it is simply a matter of arranging the control points in order and combining information where necessary.

The auxiliary function $\text{add}$ adds one state to a sequence (it inserts the new state into the sequence using the path order to determine where to do the insertion). Assume the state $m = (v, s)$ and that $\delta_0 = (w, t)$.

$$\text{add}(m, \delta) = \begin{cases} m \delta & v \leq w \\ \delta_0 \text{add}(m, \delta \geq 1) & w \leq v \\ (v, s \sqcup t) \delta \geq 1 & w = v \end{cases}$$

Now we can define $\delta^1 \cup \delta^2$ by

$$\delta^1 \cup \delta^2 = \begin{cases} \text{add}((v, s), \delta^2) & \delta^1 \equiv (v, s) \\ \delta^1 \geq 1 \cup \text{add}(\delta_0, \delta^2) & \text{otherwise} \end{cases}$$

Proposition 2.1. If $\delta^1$ and $\delta^2$ are sequences in which control points appear at most once, then $\delta^1 \subseteq \delta^1 \cup \delta^2$. Note that the $\delta^i$ may have common control points but within each sequence, control points may only appear once.

3 The Logical System: Q and PLF

Specification includes describing what the state of a program at a point ought to be (or ought not to be); verification includes the task of checking whether the specification is met. We need a language for describing properties. Section 3.1 first discusses what it means to say that a proposition about a state is true; Section 3.2 introduces the notion of a predicate that can be used to express properties of the state of variables. This is built on in Section 3.3 that describes the program logic $PLF$ that can be used to describe properties of runs of the program under study (this section deals with the scalar version of the logic — only values and program variables appear in formulas, and no symbolic expressions over values is allowed, with the full symbolic logic is introduced in Section 5). Section 3.4 concludes by examining an important subset of $PLF$.

3.1 What is truth?

Section 2.3 described how any data type could be represented as a lattice. The data type of truth values is particularly important because of the important role that this type plays. Some program variables that can have truth values (these commonly called boolean in programming languages, though we shall avoid the use of the term boolean). Second (and much more important), specifications of a program or program segment contain statements about the state at control points, and we shall want to know whether
these statements are true; what it means to say that something is true or false is central to the whole specification and verification endeavour. For these reasons, although the truth data type could be dealt with generally, it is worth while to deal with it separately.

The two most useful truth values are \( t \) (true) and \( f \) (false), and could have a number of uses as values of program variables. As truth values they also play an important role: determining whether a proposition is true or false of a particular state of the program. However, a two-valued logic system is insufficient.

First, for the general reasons discussed in Section 2.3 it is useful to have values to represent under-determined and inconsistent truth values. Because of the special role that truth values have, we represent bottom and top by \( \bot \) and \( T \) respectively. For example, a truth variable \( x \) having the \( \bot \) value, could represent the variable having an uninitialised value. (It is unlikely that a programmer would explicitly refer to the constants \( \bot \) and \( T \) in a program.)

The truth values \( \bot \) and \( T \) are also useful values for interpreting the truth of propositions about the state of the program. This can be better explained later once the logical language has been defined. Subsequent sections explain the logical language and readers should suspend judgement until Section 3.2 why a four-valued logic system is a good one to use.

In summary, the four-valued logic system \( Q = \{ \bot, f, t, T \} \) is the truth system used. \( f \) and \( t \) represent the boolean values false and true, \( \bot \) represents an unknown or neither-true-nor-false truth value, and \( T \) represents an inconsistent or both-true-and-false truth value. \( B \) denotes the set \( \{ f, t \} \) (so \( B \subset Q \)).

\( Q \) is a bilattice, a set with two partial orders. The bilattice \( Q \) has received much attention (e.g. \([5, 7, 6]\)), and is graphically shown in Figure 2. The two orderings are the truth ordering \( \preceq \) (shown graphically going from left to right) and the information ordering on the truth domain \( \preceq \) (shown graphically as going from bottom to top). (It is useful because \( Q \) is so important to use \( \preceq \) to represent the information ordering on the truth domain.)

![Figure 2: The Bilattice Q](image)

Informally, the information ordering indicates how much information the truth value contains: the minimal element \( \bot \) contains no truth information; the mutually incommensurable elements \( f \) and \( t \) contain sufficient information to determine truth exactly; and the maximal element \( T \) contains inconsistent truth information. The truth ordering indicates how true a value is. The minimum element in the ordering is \( f \) (without question not true); and the maximum element is \( t \) (without question true). The two elements \( \bot \) and \( T \) are intermediate in the ordering — in the first case, the lack of information places it between \( f \) and \( t \), and in the second case, inconsistent information does.

For representing and operating on \( Q \) as a set of truth values, there are natural definitions for negation, conjunction and disjunction, namely the weak negation operation of the bilattice and the meet and join of the \( Q \) with respect to the truth ordering \([5]\). These definitions are shown in Figure 3.

### 3.2 Expressing properties of a state

A predicate is a sentence about the state of a program at a control point. It may or may not be true, depending on what the value of the state is. Formally, a predicate is a function from the state space to the truth domain, i.e. of type \( S \rightarrow Q \). Thus, given a predicate and a state, we can determine how the predicate is true (the truth value of a predicate with respect to a state can be any of the four values in \( Q \)).

The exposition below assumes that the program under study has \( n \) variables of types \( T_1 \) to \( T_n \), and so the state space of the variables of is \( S = \text{fluff}(T_1) \times \ldots \times \text{fluff}(T_n) \) where the \( T_i \) may or may not be distinct.
The interesting predicates are monotonic: $p$ is monotonic if $s \subseteq t$ implies that $p(s) \leq p(t)$. Monotonic predicates are interesting because they capture our understanding of information orderings as abstractions. As we get more information about the state we are in, the more we know about the state, and the more we can say. Thus, the truth values of a predicate will change as we know more; but the way it can change must reflect the nature of the abstraction relation. Suppose $s \subseteq t$: this implies that $s$ is an abstraction of $t$. Now suppose that $p(s) = \perp$, then since there is more information in $t$ it could be (depending on $p$) that $p(t) = \perp$ or $p(t) = f$ or $p(t) = T$. If $p(s) = t$, then it could be that $p(t) = t$ or $T$. But it could not be that $p(t) = \perp$ since if we know more about $t$ than $s$ how could we know less about the truth of $p$ at $t$? Nor could it be that $p(t) = f$ since this would violate monotonicity. $M$ is the set of all monotonic predicates.

### 3.2.1 Justifying a four valued logic

There are two primary reasons why a two valued logic is inadequate. Suppose $p$ is a monotone predicate with $p(t_1) = t$ and $p(t_2) = f$ and $s \subseteq t_1, s \subseteq t_2$. What is $p(s)$? Recall that the $\subseteq$ relation is an abstraction relation — $s$ should share the common properties of the $t_i$ and suppress the differences. For this reason, having $p(s) = t$ or $p(s) = f$ would violate our understanding of abstraction. The only sensible alternative is to introduce and ‘don’t know’ truth value, $\perp$. A similar argument for the $T$ truth value can be made by considering a state $s'$ such that $t_1, t_2 \not\subseteq s'$.

Moreover, in the next section we shall introduce negation. We require all our predicates to be monotonic. Suppose we only had two truth values (presumably $f$ and $t$). The only sensible ordering would be to have $f < t$. But now consider the problem of negation. Suppose $p$ is monotonic and $s \subseteq s$. Hence, $p(s) < p(t)$. Fine. But $\neg p(s) \not< \neg p(t)$. Not fine.

### 3.3 The program logic PLF

This section introduces the program logic that is used to describe desired behaviour of the program. The building blocks of the logic are:

- the constant predicates $t$ and $f$;
- control predicates that can be used to ask whether the program is in a given state;
- the usual connectives such as conjunction.

The semantics of a PLF formula are given with respect to a sequence of states.

- The constant predicate $t$ ($f$) is $t$ ($f$) of every sequence.
- A control predicate is of the form $(v, p) \sim g$, where $v$ is a control point, $p$ is a predicate, and $g$ is a PLF formula. Such a control predicate asks of a sequence whether $p$ is true of the first state in the sequence which is at the control point $v$. If so, it asks whether $g$ is true of the rest of the sequence. The semantics of a control predicate also demand that there be a state in the sequence which is at the control point $v$. 

---

### Figure 3: Conjunction, disjunction and negation operators for $Q$

<table>
<thead>
<tr>
<th>$\wedge$</th>
<th>$\perp$</th>
<th>$f$</th>
<th>$t$</th>
<th>$T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\perp$</td>
<td>$\perp$</td>
<td>$f$</td>
<td>$f$</td>
<td>$f$</td>
</tr>
<tr>
<td>$f$</td>
<td>$f$</td>
<td>$f$</td>
<td>$f$</td>
<td>$f$</td>
</tr>
<tr>
<td>$t$</td>
<td>$f$</td>
<td>$t$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$f$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
</tbody>
</table>

### Truth Table for Negation

<table>
<thead>
<tr>
<th>$\neg$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\perp$</td>
</tr>
<tr>
<td>$f$</td>
</tr>
<tr>
<td>$t$</td>
</tr>
<tr>
<td>$T$</td>
</tr>
</tbody>
</table>

---

Siacsit '97
• The only connectives we need define are negation and conjunction. The other connectives can be defined as shorthand. We use \((v, p)\) as short-hand for \((v, p) \rightarrow t\). We shall also use shorthands such as \((v, p \land q)\) for \((v, p) \land (v, q)\).

Syntactically, the set of PLF formulas is given by:

\[
PLF ::= f \mid t \mid (V, \mathcal{M}) \rightarrow PLF \mid PLF \land PLF \mid \neg PLF
\]

Formally, the semantics of a formula is given by the satisfaction relation \(\text{Sat}: \mathcal{S}^\omega \times PLF \rightarrow \mathcal{Q}\) defined as follows. Let \(e\) denote the empty sequence.

**Definition 3.1.**
1. \(\text{Sat}(\epsilon, g) = \perp\);
2. \(\text{Sat}(\sigma, t) = t\);
3. \(\text{Sat}(\sigma, f) = f\);
4. \(\text{Sat}(\sigma, (v, p) \rightarrow g) = \begin{cases} p(s) \land \text{Sat}(\sigma_{\geq 1}, g) & \sigma_0 \equiv (w, s), v = w \\ \text{Sat}(\sigma_{\geq 1}, (v, p) \rightarrow g) & \sigma_0 \equiv (w, s), v \neq w \end{cases} \)
5. \(\text{Sat}(\sigma, g_1 \land g_2) = \text{Sat}(\sigma, g_1) \land \text{Sat}(\sigma, g_2)\); and
6. \(\text{Sat}(\sigma, \neg g) = \neg \text{Sat}(\sigma, g)\).

The satisfaction relation is monotonic with respect to the abstraction relation — this is a property that will be important to use later.

**Lemma 3.1.** For every PLF, \(g\), if \(q \leq \text{Sat}(\sigma, g)\) and \(\sigma \subseteq \sigma'\) then \(q \leq \text{Sat}(\sigma', g)\).

### 3.4 Trajectory formulas

We focus our attention on a subclass of PLF called the set of trajectory formulas (TF) because TF has some very pleasant properties when it comes to verification.

First, rather than allowing arbitrary monotonic predicates as the building blocks, we start with elementary predicates. The set of elementary predicates

- The constant predicate that is true everywhere. We overload the symbol \(t\) to represent this.
- The predicates that allow us to check whether a variable has a given value.
- Predicates of the form \(t \Rightarrow g\) and \(f \Rightarrow g\) where \(g\) is elementary.

This second class of predicates are of the form \([j] = r\). Informally, a predicate like this asks whether the \(j\)-th component of the current state has the value \(r\). The value \(r\) must be an element of \(T_j\) (i.e. it cannot be a top or bottom element). The formal definition is given below.

For each \(\text{fluff}(T_j)\), the function \(\overset{\Rightarrow}{=} : \text{fluff}(T_j) \times T_j \rightarrow \mathcal{Q}\) can be used to compare two elements of \(\text{fluff}(T_j)\). Informally \(x \overset{\Rightarrow}{=} y\) if \(x\) and \(y\) are the same; it is \(f\) if \(x\) and \(y\) are not comparable; it is \(\perp\) if \(x\) is \(U\) (since there is too little information to tell); and it is \(T\) if \(x\) is \(Z\) (since there is inconsistent information). Note this function is not symmetric since there is a (syntactic) restriction that the right-hand operand (in this case the \(y\) is always an element of \(T_j\)). Formally,

\[
x \overset{\Rightarrow}{=} y \overset{\text{def}}{=} \begin{cases} T & x = Z \\ t & x = y \\ f & x \not\leq y, y \not\leq x \\ \perp & x = U \end{cases}
\]

Formally, the set of all elementary predicates is denoted \(\mathcal{E}\), and is described below:
If \( p \in \mathcal{E} \), then

\[
P(s) = \begin{cases} 
  t & \text{def } t \\
  p_1(s) & \text{def } t \Rightarrow p_1 \\
  t & \text{def } f \Rightarrow p_1 \\
  s_j \vdash p & \text{def } [j] = r, s \equiv (s_1, \ldots, s_j, \ldots, s_n)
\end{cases}
\]

At present, the usefulness of formulas of the type \( t \Rightarrow p \) is very limited — but these formulas will play a very useful role later when variables are allowed. Note that by definition, elementary predicates are monotonic. Elementary predicates have the property of being simple [10], in that there is a unique weakest element in the state space for which it becomes true. If \( p \equiv [j] = r \), then \( \delta_p = (U_1, \ldots, U_{j-1}, r, U_{j+1}, \ldots, U_n) \) has the property that:

- \( p(\delta_p) = t \), moreover
- \( p(s) = t \) implies that \( \delta_p \subseteq s \) and indeed more generally
- \( t \leq p(s) \) exactly when \( \delta_p \subseteq s \).

Such an element is known as a defining element.

Sense and sensibility

A PLF formula \( g \) is sensible with respect to a program graph if

- The set of control points that appear syntactically in \( g \) is totally ordered by the path order \( \preceq \);
- In every sub-formula of the form \((g_1, g_2) \leadsto \) the control points in \( g_1 \) appear strictly before the control points in \( g_2 \) in the ordering \( \preceq \).

A PLF formula \( g \) is a trajectory formula with respect to a program graph if

- It is sensible;
- Only \( \Lambda \) and \( \leadsto \) are used as connectives;
- The underlying predicates are all elementary.

4 The basic approach to verification

4.1 Characterising sequences

In many descriptions of logics (especially modal or temporal logics), rather than defining the semantics of a formula by using a satisfaction relation, the semantics of a formula is given directly by the set of sequence or states (depending on the framework) that satisfy them. Although an elegant characterisation, it is often not a practical one, because such sets are extremely large. Such a characterisation may be appropriate in the framework of this paper, where one sequence can characterise very many sequences (all those above it in the partial order). A characterising sequence of a formula can be used effectively during the verification process.

The first result of this section is to show that every TF formula has a unique weakest (most abstract) sequence that satisfies it. If \( g \) is a TF formula, we can construct the sequence \( \delta(g) \) such that:

- \( t \leq \text{Sat}(\delta(g), g) \); and more generally
- \( \delta(g) \subseteq \sigma \iff t \leq \text{Sat}(\sigma, g) \). (Note, that here the \( \subseteq \) relation is that of Definition 2.3.)

Thus \( \sigma \) satisfies \( g \) exactly if \( \delta(g) \) is more abstract than it.
4.2 Defining sequences

Let \( g \) be a TF on a given program graph \( G = (V, E) \) with path order \( \leq \). \( \delta(g) \), the defining sequence of \( g \) is defined as follows:

\[
\delta(g) = \begin{cases} 
(v, \delta_p) & g \equiv (v, p) \\
(v, \delta_p) \delta_{g_1} & g \equiv (v, p) \leadsto g_1 \\
\delta(g_1) \sqcup \delta(g_2) & g \equiv g_1 \land g_2
\end{cases}
\]

Where there is little chance of confusion, we write \( \delta^g \) for \( \delta(g) \). Note that \( \delta^g \) will be finite for each TF, \( g \).

**Lemma 4.1.** For every TF \( g \), \( \delta^g \subseteq \sigma \) implies that \( t \leq \text{Sat}(\sigma, g) \).

**Lemma 4.2.** For every TF \( g \), \( t \leq \text{Sat}(\sigma, g) \) implies that \( \delta^g \subseteq \sigma \).

And putting these results together.

**Theorem 4.3.** For every TF \( g \) on a given program graph the set of sequences that satisfy it are exactly those that are related to \( \delta^g \) by the partial order:

\[
t \leq \text{Sat}(\sigma, g) \iff \delta^g \subseteq \sigma
\]

The importance of this result is that every trajectory formula has a unique minimal sequence that satisfies it.

4.3 Defining trajectories

The previous section looked at defining sequences that characterise a formula. Of course, not all sequences can be produced by the program graph — they are not all trajectories. This section shows that subject to certain conditions, trajectory formulas also have defining trajectories, i.e. unique minimal sequences that satisfy them.

The first part of this section shows how to take a sequence and find the minimal trajectory for which it is an abstraction.

Let \( \delta \) be a sequence of states. We define the function \( \tau : S^\omega \rightarrow S^\omega \) below. Let \( \tau = \tau(\delta) \) and assume that \( \tau' \) is any trajectory such that \( \delta \subseteq \tau' \). In the construction of \( \tau \) we shall use the information in \( \delta \). To accomplish this, we shall use a sequence of suffixes of \( \delta \), denoted \( \delta^0, \delta^1, \ldots \) (the elements in this sequence may not all be distinct). For convenience, we assume that \( \delta^0 = (v, s) \) and that \( \gamma(\tau_i) = (w, t) \).

\[
(\tau_0, \delta^0) = (\delta_0, \delta_{\geq 1}) \\
(\tau_{i+1}, \delta^{i+1}) = \begin{cases} 
(\gamma(\tau_i), \delta^i) & v \neq w \\
((v, s \sqcup t), \delta^i_{\geq 1}) & v = w
\end{cases}
\]

Where there is no chance of confusion, we write \( \tau^g \) for \( \tau(g) \). \( \tau^g \) is called the defining trajectory of \( g \).

**Lemma 4.4.** \( \delta \subseteq \tau(\delta) \)

The next result is analogous to Theorem 4.3 and shows that the defining trajectory of a formula characterises the formula.

**Proposition 4.5.** For every \( g \in \text{TF} \) on a given program graph the set of trajectories that satisfy it are exactly those that are related to \( \tau^g \) by the partial order:

\[
t \leq \text{Sat}(\sigma, g) \iff \tau^g \subseteq \sigma
\]
4.4 Assertions

The program logic PLF is used as the basis for specifying properties. In this framework a property is expressed as a precondition/postcondition pair called an assertion.

Definition 4.1. A program satisfies the assertion $(g \Rightarrow h)$ if every trajectory of the program that satisfies $g$ also satisfies $h$:

$$\forall \text{ trajectories } \sigma, t \subseteq \text{Sat}(\sigma, g) \implies t \subseteq \text{Sat}(\sigma, h)$$

The results of the previous section lay the basis for verifying an assertion. Suppose we have two trajectory formulas $g$ and $h$ and we wish to show that $(g \Rightarrow h)$. If it's the case that $\delta^g \subseteq \tau^g$ we can conclude the result is true. For suppose a trajectory $\sigma$ satisfies $g$, then we know that $\tau^g \subseteq \sigma$. So, by transitivity $\delta^h \subseteq \sigma$. But this means that $\sigma$ satisfies $h$. The converse also holds.

Theorem 4.6. If $g, h \in TF$, $(g \Rightarrow h) \iff \delta^h \subseteq \tau^g$.

A more general result is:

Theorem 4.7. If $g \in TF$, $(g \Rightarrow h) \iff t \subseteq \text{Sat}(\tau^g, h)$.

5 The Symbolic Logic

The logic PLF described so far is relatively restrictive because it only allows reference to scalar values. This means that expressions can only contain values of the appropriate type; the weakness of this the lack of expressiveness of the logic as many assertions may be needed to specify the required behaviour of the program. By including ‘variables’ in the logic we are able to quantify implicitly over many values at the same time.

Before proceeding, we need to choose terminology carefully as ‘variable’ is a term that can have several meanings. In this context, a variable is a variable of the programming language, and a symbolic constant is a ‘variable’ of the logic. Symbolic constants are meta-variables: they do not appear as part of the program text but are used for reasoning and for a symbolic computation of the program graph.

Example 5.1. Suppose we have (program) variables first, second and sum and symbolic constants $x$ and $y$. A possible symbolic PLF formula is $(v, \text{first} = x \land \text{second} = y)$. A possible symbolic assertion is

$$(v, \text{first} = x \land \text{second} = y) \Rightarrow (w, \text{sum} = x + y).$$

Informally, this assertion says that in every trajectory of the program in which the program reaches control point $v$ for all values $x$ such that $\text{first} = x$ and for all values $y$ such that $\text{second} = y$, the program will reach control point $w$ with $\text{sum} = x + y$. (When the program gets to point $w$, $\text{sum}$ will have the sum of the values that the variables $\text{first}$ and $\text{second}$ had at point $v$.)

5.1 Symbolic expressions

There is a set of symbolic constants $S_C$; each symbolic constant is typed, and each PLF must be properly typed with respect to the program (this should be easily checked by a tool). An interpretation is a mapping that gives each symbolic constant a value. Each interpretation is of type $S_C \rightarrow UT_i$. The set of all interpretations is denoted $\Phi$.

Each type $T_i$ has a functions defined on it (for example, the integer types have operations like addition and multiplication defined on it), and we allow expressions over types to include appropriate variables. An interpretation of symbolic constants can be extended to an interpretation of expressions in the standard way. Essentially we have a multi-sorted $\Sigma$ algebra (the sorts being the types and PLF) [12].
Example 5.2. Suppose we have two variables (x of type integer, and z of type string) and two symbolic constants (a and b of type integer), and the symbolic PLF formula

\[ g \, \text{def} \, (v, x = a) \land (w, z = a > b). \]

Given the interpretation \( \phi \in \Phi \), defined by \( \{ a \leftrightarrow 3, b \leftrightarrow 4 \} \), we can find the interpretation of \( g \) with respect to \( \phi \) (denoted \( \phi(g) \)) by substituting in the values into \( g \) for the symbolic constants to get

\[ \phi(g) = (v, x = 3) \land (w, z = f). \]

Each data type, \( T \), is allowed to have expressions of the form \( b \rightarrow t \), where \( t \) is an expression of type \( T \) and \( b \) is an expression of type \( Q \). We define:

\[
\phi(b \rightarrow t) \, \text{def} \, \begin{cases} 
\phi(t) & \text{if } \phi(b) = t \\
U_T & \text{if } \phi(b) = f \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

The expression \( b \rightarrow t_1 | t_2 \) is short-hand for \( b \rightarrow t_1 \lor \lnot b \rightarrow t_2 \).

5.2 Symbolic states

Recall that each state contains two components: a control point and a data part (the values of the variables represented as an \( n \)-tuple). The data part can be represented symbolically by an \( n \)-tuple of symbolic expressions of the appropriate type. For the control part we need to take more care.

Definition 5.1. A symbolic state is a set \( \{ b_1 \leftrightarrow (v_1, s_1), \ldots, b_m \leftrightarrow (v_m, s_m) \} \) where the \( b_i \) are symbolic \( Q \)-expressions and the \( s_i \) are symbolic data parts of states and we make the following restriction:

- for each interpretation at most one component has a guard that is true (\( \forall \phi \in \Phi, \forall i, j, i \neq j \implies \phi(b_i \land b_j) = f \)).

Thus, a symbolic state is a list of guard, state pairs such that each interpretation makes at most one of the guards true, which makes the definition below of an interpretation of a symbolic state well defined:

If \( S = \{ b_1 \leftrightarrow (v_1, s_1), \ldots, b_m \leftrightarrow (v_m, s_m) \} \),

\[ \phi(S) \, \text{def} \, \begin{cases} 
(v_i, \phi(s_i)) & \exists b_i \exists \phi(b_i) = t \\
(\text{undef}, U) & \text{otherwise}
\end{cases} \]

Each element of a symbolic state is called a component.

The symbolic next state function

We extend the next state function in the following way. First, for each component, we define

\[ Y(b \rightarrow (v, s)) = \{ b \land b'(s) \rightarrow (w, s') : b \land b'(s) \neq f, \exists (v, w, b') \in E, s' = (\text{op } v)s \}, \]

and

\[ Y(S) = \bigcup_{\gamma \in S} Y(\gamma). \]

\( Y(S) \) captures the next state function's behaviour, but may not be in a canonical form. For example, we may have both \( b_1 \leftrightarrow (v, s_1) \) and \( b_2 \leftrightarrow (v, s_2) \) in \( Y(S) \). From an algorithmic point of view it may be sensible to replace these two components with a single component that contains semantically equivalent information, but this does not affect the theoretical basis laid down here.
5.3 Symbolic assertions

A symbolic assertion is an assertion containing symbolic PLF formulas. Scalar assertions are either true or false. A symbolic assertion tells us for which interpretations the assertion is true.

Definition 5.2. \( (g \Rightarrow h)^{\text{def}} \{ \phi \in \Phi : (\phi(g) \Rightarrow \phi(h)) \} \).

(Note that the assertion on the left is a symbolic assertion, while the assertion on the right is the scalar assertion defined in Definition 4.1). If a symbolic assertion is true for all interpretations, it is denoted thus: \( \models (g \Rightarrow h) \).

Definition 5.3. \( \models (g \Rightarrow h)^{\text{def}} \Phi = \{ \phi \in \Phi : (\phi(g) \Rightarrow \phi(h)) \} \).

5.4 Verifying symbolic assertions

In this section, we extend the results of Section 4 to show how symbolic assertions of a certain type can be verified.

Definition 5.4. If \( p \) is a symbolic predicate, it is monotone if \( \forall \phi \in \Phi, \phi(p) \in M \). \( \delta_p \), a symbolic expression over \( S \), is a defining element for a symbolic predicate \( p \) if \( \forall \phi \in \Phi, \phi(\delta_p) \) is a defining element for \( \phi(p) \).

The set of symbolic elementary formulas \( E ::= t | \vec{B} \Rightarrow E | | N | = \vec{T} \), where \( \vec{B} \) are the symbolic expressions over \( B \) and \( \vec{T} \) are symbolic expressions over \( T \) where \( T \) is the appropriate type for the component mentioned.

Proposition 5.1. If \( p \in E \), then

\[ \delta_p = \begin{cases} \{ \delta_{p_1} \} & \text{if } p \overset{\text{def}}{=} t \\ \{ b \leftarrow \delta_{p_1} \} & \text{if } p \overset{\text{def}}{=} b \Rightarrow p_1 \\ \{ (U_1, \ldots, U_{j-1}, r, U_{j+1}, \ldots, U_n) \} & \text{if } p \overset{\text{def}}{=} [j] = r. \end{cases} \]

Practical implementation of symbolic sequences is critical in the development of this theory.

6 Practical verification

The previous sections outlines a methodology for verifying assertions about software using symbolic trajectory evaluation. This will form the computational heart of the verification process. However, it will not be enough to form a practical methodology. A practical verification methodology is outside the scope of this paper, but will rely on the points discussed below.

6.1 An integrated tool

It is a basic assumption of this research that a practical verification methodology requires a good integrated tool.

- Verification should be part of the design and coding process, not an ex post facto event.
- Hand proofs are tedious and error-prone.
- Due to the computational complexity of the task, verification will consist of a number of different tasks. To be usable, a tool is necessary to make the process easier and to facilitate management of the proof task.
- Tools should be designed to use by programmers, not by verification experts.
6.2 Powerful reasoning and data structure facilities

Verification will require reasoning about the data which programs manipulate. A simple example of such reasoning is determining whether \( x + y = y + x \) (which may or may not be the case depending on the types of \( x \) and \( y \) and the meaning of \( '+' \)). The tool discussed above will need to do this, and there are several strategies that can be adopted.

- The use of data structures such as ordered binary decision diagrams and binary moment diagrams [1, 2]. Using these data structures has qualitatively changed the type of hardware system that is feasible to verify.
- Libraries to represent domain knowledge.
- Interface to general theorem proving tools

6.3 Compositional Theory

Symbolic trajectory evaluation is an automatic verification method and thus subject to limitations. Some of these limitations are due to the limitations of the algorithms used to implement the theory, and some due to the inherent computational complexity of the problem of verification. A compositional theory for verification can overcome some of these limitations. The idea is that STE is used on relatively small pieces of code where the level of tedium is high, and the compositional theory can be use at a higher level, where human insight can be used effectively. This strategy has been used effectively in hardware verification [9]. In addition, a compositional approach is useful when verification is part of an ongoing process and the parts of the program are verified as the program is developed.

6.4 Abstraction

The program graph together with the associated next state function is a mathematical model of the program. It is possible to build abstract models of the program that have the property that if an assertion is true of the abstraction, it is true of the underlying model. The idea of using abstraction is well known in the verification community (see e.g. [4]) and has been successfully used. The data model of STE (the use of lattice state spaces) naturally supports abstraction; however, it is possible to build and use abstractions in different ways [8]. In the context of software verification, this would correspond to control abstraction. One possibility is to partially automate the abstraction building process.

The use of compositionality and abstraction will be critical for the efficient verification of software containing procedures and loops.

7 Conclusion

This paper has presented a framework for the verification of software based on the theory of symbolic trajectory evaluation. The basic idea is that programs are represented by program graphs (essentially extended finite state machines), and data is represented by partially-ordered state spaces. The specification is given by using a set of assertions written with a simple program logic (PLF). Many PLF formulas have simple characterising sequences that can be used in the verification process.

Practical verification algorithms and tools based on this theory are sketched. The positive experience in verifying hardware [10] indicates that work in developing these algorithms and tools are worth pursuing.

Two thrusts of work must be done:

- Extending and filling out the theoretical framework. This involves developing the theory of compositionality and abstraction, as well extending the basic theory as the current theory is heavily dependant on the use of trajectory evaluation.
- Building tools and gathering experimental evidence.
References


