PROCEEDINGS / KONGRESOPSPOMMINGS

6th
SOUTHERN AFRICAN COMPUTER SYMPOSIUM

6de
SUIDELIKE-AFRIKAANSE REKENAARSIMPOSIUM

De Overberger Hotel, Caledon
2 - 3 JULY 1991

SPONSORED by
ISM
FRD
GENMIN

EDITED by
M H LINCK
Department of Computer Science
University of Cape Town
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword</td>
<td>1</td>
</tr>
<tr>
<td>Organising Committee</td>
<td>2</td>
</tr>
<tr>
<td>Referees</td>
<td>3</td>
</tr>
<tr>
<td>Program</td>
<td>5</td>
</tr>
<tr>
<td>Papers (In order of presentation)</td>
<td>9</td>
</tr>
<tr>
<td>&quot;A value can belong to many types&quot; B H Venter, University of Fort Hare</td>
<td>10</td>
</tr>
<tr>
<td>&quot;A Transputer Based Embedded Controller Development System&quot; M R Webster, R G Harley, D C Levy &amp; D R Woodward, University of Natal</td>
<td>16</td>
</tr>
<tr>
<td>&quot;Improving a Control and Sequencing Language&quot; G Smit &amp; C Fair, University of Cape Town</td>
<td>25</td>
</tr>
<tr>
<td>&quot;Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers&quot; P Machanick, University of Witwatersrand</td>
<td>40</td>
</tr>
<tr>
<td>&quot;Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface&quot; L van Zijl &amp; D Mitton, University of Stellenbosch</td>
<td>51</td>
</tr>
<tr>
<td>&quot;Product Form Solutions for Multiserver Centres with Hierarchical Classes of Customers&quot; A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig</td>
<td>69</td>
</tr>
<tr>
<td>&quot;A Reusable Kernel for the Development of Control Software&quot; W Fouché and P de Villiers, University of Stellenbosch</td>
<td>83</td>
</tr>
<tr>
<td>&quot;An Implementation of Linda Tuple Space under the Helios Operating System&quot; P G Clayton, E P Wentworth, G C Wells and F de Heer-Menlah, Rhodes University</td>
<td>95</td>
</tr>
<tr>
<td>&quot;The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System&quot; K Macgregor, University of Cape Town &amp; R Campbell University of Illinois at Urbana-Champaign</td>
<td>107</td>
</tr>
</tbody>
</table>
"Concurrency Control Mechanisms for Multidatabase Systems"
A Deacon, University of Stellenbosch  118

"Extending Local Recovery Techniques for Distributed Databases"
H L Victor & M H Rennhackkamp, University of Stellenbosch  135

"Analysing Routing Strategies in Sporadic Networks"
S Melville, University of Natal  148

The Design of a Speech Synthesis System for Afrikaans"
M J Wagener, University of Port Elizabeth  167

"Expert Systems for Management Control: A Multiexpert Architecture"
V Ram, University of Natal  177

"Integrating Similarity-Based and Explanation-Based Learning"
G D Oosthuizen and C Avenant, University of Pretoria  187

"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town  201

"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch  211

"Building a secure database using self-protecting objects"
M Olivier and S H von Solms, Rand Afrikaans University  228

"Modelling the Algebra of Weakest Preconditions"
C Brink and I Rewitsky, University of Cape Town  242

"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch  262

"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal  276
FOREWORD

The 6th Computer Symposium, organised under the auspices of SAICS, carries on the tradition of providing an opportunity for the South African scientific computing community to present research material to their peers.

It was heartening that 31 papers were offered for consideration. As before all these papers were refereed. Thereafter a selection committee chose 21 for presentation at the Symposium.

Several new dimensions are present in the 1991 symposium:

* The Symposium has been arranged for the day immediately after the SACLA conference.

* It is being run over only 1 day in contrast to the 2-3 days of previous symposia.

* I believe that it is first time that a Symposium has been held outside of the Transvaal.

* Over 85 people will be attending. Nearly all will have attended both events.

* A Sponsorship package for both SACLA and the Research Symposium was obtained. (This led to reduced hotel costs compared to previous symposia)

A major expense is the production of the Proceedings of the Symposium. To ensure financial soundness authors have had to pay the page charge of R20 per page.

A thought for the future would be consideration of a poster session at the Symposium. This could provide an alternative approach to presenting ideas or work.

I would sincerely hope that the twinning of SACLA and the Research Symposium is considered successful enough for this combination survive. As to whether a Research Symposium should be run each year after SACLA, or only every second year, is a matter of need and taste.

A challenge for the future is to encourage an even greater number of MSc & PhD students to attend the Symposium. Unlike this year, I would recommend that they be accommodated at the same cost as everyone else. Only if it is financially necessary should the sponsored number of students be limited.

I would like to thank the other members of the organising committee and my colleagues at UCT for all the help that they have given me. A special word of thanks goes to Prof. Pieter Kritzinger who has provided me with invaluable help and ideas throughout the organisation of this 6th Research Symposium.

M H Linck
Symposium Chairman
SYMPOSIUM CHAIRMAN

M H Linck, University of Cape Town

ORGANISING COMMITTEE

D Kourie, Pretoria University.
P S Kritzinger, University of Cape Town.
M H Linck, University of Cape Town.

SPONSORS

ISM
GENMIN
FRD
### LIST OF REFEREES FOR 6th RESEARCH SYMPOSIUM

<table>
<thead>
<tr>
<th>NAME</th>
<th>INSTITUTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnard, E</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Becker, Ronnie</td>
<td>UCT</td>
</tr>
<tr>
<td>Berman S</td>
<td>UCT</td>
</tr>
<tr>
<td>Bishop, Judy</td>
<td>Wits</td>
</tr>
<tr>
<td>Berman, Sonia</td>
<td>UCT</td>
</tr>
<tr>
<td>Brink, Chris</td>
<td>UCT</td>
</tr>
<tr>
<td>Bodde, Ryn</td>
<td>Networks Systems</td>
</tr>
<tr>
<td>Bornman, Chris</td>
<td>UNISA</td>
</tr>
<tr>
<td>Bruwer, Piet</td>
<td>UOFS</td>
</tr>
<tr>
<td>Cherenack, Paul</td>
<td>UCT</td>
</tr>
<tr>
<td>Cook Donald</td>
<td>UCT</td>
</tr>
<tr>
<td>de Jaeger, Gerhard</td>
<td>UCT</td>
</tr>
<tr>
<td>de Villiers, Pieter</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Ehlers, Elize</td>
<td>RAU</td>
</tr>
<tr>
<td>Eloff, Jan</td>
<td>RAU</td>
</tr>
<tr>
<td>Finnie, Gavin</td>
<td>Natal</td>
</tr>
<tr>
<td>Gaynor, N</td>
<td>AECI</td>
</tr>
<tr>
<td>Hutchinson, Andrew</td>
<td>UCT</td>
</tr>
<tr>
<td>Jourdan, D</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kourie Derrick</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kritzinger, Pieter</td>
<td>UCT</td>
</tr>
<tr>
<td>Krzesinski, Tony</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Laing, Doug</td>
<td>ISM</td>
</tr>
<tr>
<td>Labuschagne, Willem</td>
<td>UNISA</td>
</tr>
<tr>
<td>Levy, Dave</td>
<td>Natal</td>
</tr>
<tr>
<td>Name</td>
<td>Institution</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>MacGregor, Ken</td>
<td>UCT</td>
</tr>
<tr>
<td>Machanick, Philip</td>
<td>Wits</td>
</tr>
<tr>
<td>Mattison Keith</td>
<td>UCT</td>
</tr>
<tr>
<td>Messerschmidt, Hans</td>
<td>UOFS</td>
</tr>
<tr>
<td>Mutch, Laurie</td>
<td>Shell</td>
</tr>
<tr>
<td>Neishlos, N</td>
<td>Wits</td>
</tr>
<tr>
<td>Oosthuizen, Deon</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Peters Joseph</td>
<td>Simon Fraser</td>
</tr>
<tr>
<td>Ram, V</td>
<td>Natal, Pmb.</td>
</tr>
<tr>
<td>Postma, Stef</td>
<td>Natal, Pmb</td>
</tr>
<tr>
<td>Rennhackkamp, Martin</td>
<td>Stellenbœsch</td>
</tr>
<tr>
<td>Shochot, John</td>
<td>Wits</td>
</tr>
<tr>
<td>Silverberg, Roger</td>
<td>Council for Mineral Technology</td>
</tr>
<tr>
<td>Smit, Riel</td>
<td>UCT</td>
</tr>
<tr>
<td>Smith, Dereck</td>
<td>UCT</td>
</tr>
<tr>
<td>Terry, Pat</td>
<td>Rhodes</td>
</tr>
<tr>
<td>van den Heever, Roelf</td>
<td>UP</td>
</tr>
<tr>
<td>van Zijl, Lynette</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Venter, Herman</td>
<td>Fort Hare</td>
</tr>
<tr>
<td>Victor, Hema</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>von Solms, Basie</td>
<td>RAU</td>
</tr>
<tr>
<td>Wagenaar, M</td>
<td>UPE</td>
</tr>
<tr>
<td>Wentworth, Peter</td>
<td>Rhodes</td>
</tr>
<tr>
<td>Wheeler, Graham</td>
<td>UCT</td>
</tr>
<tr>
<td>Wood, Peter</td>
<td>UCT</td>
</tr>
</tbody>
</table>
6TH RESEARCH SYMPOSIUM - 1991

FINAL PROGRAM

TUESDAY 2nd July 1991

10h00 - 13h00  Registration
13h00 - 13h50  PUB LUNCH

14h00 - 15h30  SESSION 1A

Venue: Hassner
Chairman: Prof Basie von Solms

14h00 - 14h30  "A value can belong to many types."
B H Venter, University of Fort Hare

14h30 - 15h00  "A Transputer Based Embedded Controller Development System"
M R Webster, R G Harley, D C Levy & D R Woodward, University of Natal

15h00 - 15h30  "Improving a Control and Sequencing Language"
G Smit and C Fair, University of Cape Town

15h30 - 16h00  TEA

SESSION 1B

Venue: Hassner C
Chairman: Prof Roelf v d Heever

14h00 - 14h30  "Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers"
P Machanick, University of Witwatersrand

14h30 - 15h00  "Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface"
L van Zijl & D Mitton, University of Stellenbosch

15h00 - 15h30  "Product Form Solutions for Multiserver Centres with Hierarchical Classes of Customers"
A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig

5
16h00 - 17h30 SESSION 2A

Venue: Hassner

Chairman: Prof Derrick Kourie

16h00 - 16h30
"A Reusable Kernel for the Development of Control Software" W Fouché and P de Villiers, University of Stellenbosch

16h30 - 17h00
"An Implementation of Linda Tuple Space under the Helios Operating System" P G Clayton, E P Wentworth, G C Wells and F de-Heer-Menlah, Rhodes University

17h00 - 17h30
"The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System" K MacGregor, University of Cape Town & R Campbell, University of Illinois at Urbana-Champaign

19h30 PRE-DINNER DRINKS

20h00 GALA CAPE DINNER
(Men: Jackets & ties)
WEDNESDAY 3rd July 1991

7h00 - 8h15 BREAKFAST

8h15 - 9h45 SESSION 3A

Venue: Hassner

Chairman: Assoc Prof P Wood

8h15 - 8h45
"Concurrency Control Mechanisms for Multidatabase Systems" A Deacon, University of Stellenbosch

8h45 - 9h15
"Extending Local Recovery Techniques for Distributed Databases" H L Victor & M H Rennhackkamp, University of Stellenbosch

9h15 - 9h45
"Analysing Routing Strategies in Sporadic Networks" S Melville, University of Natal

9h45 - 10h15 TEA

10h15 - 11h00 SESSION 4

Venue: Hassner

Chairman: Prof P S Kritzinger
Invited paper: E Coffman

11h00 - 11h10 BREAK

SESSION 3B

Venue: Hassner C

Chairman: Prof G Finnie

8h15 - 8h45
"The Design of a Speech Synthesis System for Afrikaans" M J Wagener, University of Port Elizabeth

8h45 - 9h15
"Expert Systems for Management Control: A Multiexpert Architecture" V Ram, University of Natal

9h15 - 9h45
"Integrating Similarity-Based and Explanation-Based Learning" G D Oosthuizen and C Avenant, University of Pretoria
11h10 - 12h40 SESSION 5A
Venue: Hassner
Chairman: Prof C Bornman

11h10 - 11h40
"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town

11h40 - 12h10
"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch

12h10 - 12h40
"Building a secure database using self-protecting objects"
M Olivier and S H von Solms, Rand Afrikaans University

SESSION 5B
Venue: Hassner C
Chairman: Prof A Krzesinski

11h10 - 11h40
"Modelling the Algebra of Weakest Preconditions"
C Brink & I Rewitsky, University of Cape Town

11h40 - 12h10
"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch

12h10 - 12h40
"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal

12h45-12h55 GENERAL MEETING of RESEARCH SYMPOSIUM ATTENDEES
Venue: Hassner
Chairman: Dr M H Linck

13h00 - 14h00 LUNCH

FINIS 6th COMPUTER SYMPOSIUM
PAPERS

of the

6TH RESEARCH SYMPOSIUM
FOREWORD

The 6th Computer Symposium, organised under the auspices of SAICS, carries on the tradition of providing an opportunity for the South African scientific computing community to present research material to their peers.

It was heartening that 31 papers were offered for consideration. As before all these papers were refereed. Thereafter a selection committee chose 21 for presentation at the Symposium.

Several new dimensions are present in the 1991 symposium:

* The Symposium has been arranged for the day immediately after the SACLAC conference.

* It is being run over only 1 day in contrast to the 2-3 days of previous symposia.

* I believe that it is first time that a Symposium has been held outside of the Transvaal.

* Over 85 people will be attending. Nearly all will have attended both events.

* A Sponsorship package for both SACLAC and the Research Symposium was obtained. (This led to reduced hotel costs compared to previous symposia)

A major expense is the production of the Proceedings of the Symposium. To ensure financial soundness authors have had to pay the page charge of R20 per page.

A thought for the future would be consideration of a poster session at the Symposium. This could provide an alternative approach to presenting ideas or work.

I would sincerely hope that the twinning of SACLAC and the Research Symposium is considered successful enough for this combination survive. As to whether a Research Symposium should be run each year after SACLAC, or only every second year, is a matter of need and taste.

A challenge for the future is to encourage an even greater number of MSc & PhD students to attend the Symposium. Unlike this year, I would recommend that they be accommodated at the same cost as everyone else. Only if it is financially necessary should the sponsored number of students be limited.

I would like to thank the other members of the organising committee and my colleagues at UCT for all the help that they have given me. A special word of thanks goes to Prof. Pieter Kritzinger who has provided me with invaluable help and ideas throughout the organisation of this 6th Research Symposium.

M H Linck
Symposium Chairman
SYMPOSIUM CHAIRMAN

M H Linck, University of Cape Town

ORGANISING COMMITTEE

D Kourie, Pretoria University.
P S Kritzinger, University of Cape Town.
M H Linck, University of Cape Town.

SPONSORS

ISM
GENMIN
FRD
# List of Referees for 6th Research Symposium

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnard, E</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Becker, Ronnie</td>
<td>UCT</td>
</tr>
<tr>
<td>Berman S</td>
<td>UCT</td>
</tr>
<tr>
<td>Bishop, Judy</td>
<td>Wits</td>
</tr>
<tr>
<td>Berman, Sonia</td>
<td>UCT</td>
</tr>
<tr>
<td>Brink, Chris</td>
<td>UCT</td>
</tr>
<tr>
<td>Bodde, Ryn</td>
<td>Networks Systems</td>
</tr>
<tr>
<td>Bornman, Chris</td>
<td>UNISA</td>
</tr>
<tr>
<td>Bruwer, Piet</td>
<td>UOFS</td>
</tr>
<tr>
<td>Cherenack, Paul</td>
<td>UCT</td>
</tr>
<tr>
<td>Cook Donald</td>
<td>UCT</td>
</tr>
<tr>
<td>de Jaeger, Gerhard</td>
<td>UCT</td>
</tr>
<tr>
<td>de Villiers, Pieter</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Ehlers, Elize</td>
<td>RAU</td>
</tr>
<tr>
<td>Eloff, Jan</td>
<td>RAU</td>
</tr>
<tr>
<td>Finnie, Gavin</td>
<td>Natal</td>
</tr>
<tr>
<td>Gaynor, N</td>
<td>AECI</td>
</tr>
<tr>
<td>Hutchinson, Andrew</td>
<td>UCT</td>
</tr>
<tr>
<td>Jourdan, D</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kourie Derrick</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Kritzinger, Pieter</td>
<td>UCT</td>
</tr>
<tr>
<td>Krzesinski, Tony</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Laing, Doug</td>
<td>ISM</td>
</tr>
<tr>
<td>Labuschagne, Willem</td>
<td>UNISA</td>
</tr>
<tr>
<td>Levy, Dave</td>
<td>Natal</td>
</tr>
</tbody>
</table>

3
<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacGregor, Ken</td>
<td>UCT</td>
</tr>
<tr>
<td>Machanick, Philip</td>
<td>Wits</td>
</tr>
<tr>
<td>Mattison Keith</td>
<td>UCT</td>
</tr>
<tr>
<td>Messerschmidt, Hans</td>
<td>UOFSS</td>
</tr>
<tr>
<td>Mutch, Laurie</td>
<td>Shell</td>
</tr>
<tr>
<td>Neishlos, N</td>
<td>Wits</td>
</tr>
<tr>
<td>Oosthuizen, Deon</td>
<td>Pretoria</td>
</tr>
<tr>
<td>Peters Joseph</td>
<td>Simon Fraser</td>
</tr>
<tr>
<td>Ram, V</td>
<td>Natal, Pmb.</td>
</tr>
<tr>
<td>Postma, Stef</td>
<td>Natal, Pmb</td>
</tr>
<tr>
<td>Rennhackkamp, Martin</td>
<td>Stellenbösch</td>
</tr>
<tr>
<td>Shochot, John</td>
<td>Wits</td>
</tr>
<tr>
<td>Silverberg, Roger</td>
<td>Council for Mineral Technology</td>
</tr>
<tr>
<td>Smit, Riel</td>
<td>UCT</td>
</tr>
<tr>
<td>Smith, Dereck</td>
<td>UCT</td>
</tr>
<tr>
<td>Terry, Pat</td>
<td>Rhodes</td>
</tr>
<tr>
<td>van den Heever, Roelf</td>
<td>UP</td>
</tr>
<tr>
<td>van Zijl, Lynette</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>Venter, Herman</td>
<td>Fort Hare</td>
</tr>
<tr>
<td>Victor, Herma</td>
<td>Stellenbosch</td>
</tr>
<tr>
<td>von Solms, Basie</td>
<td>RAU</td>
</tr>
<tr>
<td>Wagenaar, M</td>
<td>UPE</td>
</tr>
<tr>
<td>Wentworth, Peter</td>
<td>Rhodes</td>
</tr>
<tr>
<td>Wheeler, Graham</td>
<td>UCT</td>
</tr>
<tr>
<td>Wood, Peter</td>
<td>UCT</td>
</tr>
</tbody>
</table>
6TH RESEARCH SYMPOSIUM - 1991

FINAL PROGRAM

TUESDAY 2nd July 1991

10h00 - 13h00  Registration
13h00 - 13h50  PUB LUNCH

14h00 - 15h30  SESSION 1A

Venue: Hassner
Chairman: Prof Basie von Solms

14h00 - 14h30
"A value can belong to many types."
B H Venter, University of Fort Hare

14h30 - 15h00
"A Transputer Based Embedded Controller Development System"
M R Webster, R G Harley, D C Levy & D R Woodward, University of Natal

15h00 - 15h30
"Improving a Control and Sequencing Language"
G Smit and C Fair, University of Cape Town

SESSION 1B

Venue: Hassner C
Chairman: Prof Roelf v d Heever

14h00 - 14h30
"Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers" P Machanick, University of Witwatersrand

14h30 - 15h00
"Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface" L van Zijl & D Mitton, University of Stellenbosch

15h00 - 15h30
"Product Form Solutions for Multiserver Centres with Hierarchical Classes of Customers" A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig

15h30 - 16h00  TEA
16h00 - 17h30  SESSION 2A

Venue: Hassner

Chairman: Prof Derrick Kourie

16h00 - 16h30
"A Reusable Kernel for the Development of Control Software" W Fouché and P de Villiers, University of Stellenbosch

16h30 - 17h00
"An Implementation of Linda Tuple Space under the Helios Operating System" P G Clayton, E P Wentworth, G C Wells and F de-Heer-Menlah, Rhodes University

17h00 - 17h30
"The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System" K MacGregor, University of Cape Town & R Campbell, University of Illinois at Urbana-Champaign

19h30          PRE-DINNER DRINKS

20h00          GALA CAPE DINNER
(Men: Jackets & ties)
WEDNESDAY 3rd July 1991

7h00 - 8h15  BREAKFAST

8h15 - 9h45 SESSION 3A

Venue: Hassner

Chairman: Assoc Prof P Wood

8h15 - 8h45
"Concurrency Control Mechanisms for Multidatabase Systems" A Deacon,
University of Stellenbosch

8h45 - 9h15
"Extending Local Recovery Techniques for Distributed Databases" H L Victor
& M H Rennhackkamp, University of Stellenbosch

9h15 - 9h45
"Analysing Routing Strategies in Sporadic Networks" S Melville,
University of Natal

SESSION 3B

Venue: Hassner C

Chairman: Prof G Finnie

8h15 - 8h45
"The Design of a Speech Synthesis System for Afrikaans" M J Wagener,
University of Port Elizabeth

8h45 - 9h15
"Expert Systems for Management Control: A Multieexpert Architecture"
V Ram, University of Natal

9h15 - 9h45
"Integrating Similarity-Based and Explanation-Based Learning"
G D Oosthuizen and C Avenant,
University of Pretoria

9h45 - 10h15  TEA

10h15 - 11h00  SESSION 4

Venue: Hassner

Chairman: Prof P S Kritzinger
Invited paper: E Coffman

11h00 - 11h10  BREAK
11h10 - 12h40 SESSION 5A

Venue: Hassner

Chairman: Prof C Bornman

11h10 - 11h40
"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town

11h40 - 12h10
"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch

12h10 - 12h40
"Building a secure database using self-protecting objects" M Olivier and S H von Solms, Rand Afrikaans University

SESSION 5B

Venue: Hassner C

Chairman: Prof A Krzesinski

11h10 - 11h40
"Modelling the Algebra of Weakest Preconditions"
C Brink & I Rewitsky, University of Cape Town

11h40 - 12h10
"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch

12h10 - 12h40
"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal

12h45-12h55 GENERAL MEETING of RESEARCH SYMPOSIUM ATTENDEES

Venue: Hassner

Chairman: Dr M H Linck

13h00 - 14h00 LUNCH

FINIS 6th COMPUTER SYMPOSIUM
PAPERS

of the

6TH RESEARCH SYMPOSIUM
A value can belong to many types

B H Venter
Department of Computer Science, University of Fort Hare
Private Bag X1314, Alice, Republic of Ciskei, Southern Africa
e-mail: cssb_v1@ufh.ac.za

Abstract

Most statically typed Algol-style languages were designed to conform with the view that a value belongs to one and only one type. The adoption of this view bought compiler simplicity at the expense of language expressivity. This trade off has been partially offset by the introduction of subtypes. The adoption of the view that types may intersect in arbitrary ways, on the other hand, requires complicated compilers or costly run-time type checks. However, language expressivity is enhanced, language semantics are simplified and object-oriented and database concepts can be integrated smoothly with the Algol style.

Keywords: Data abstraction, object-oriented programming, polymorphism, type checking
Computing Reviews Categories: D.3.3

1 Introduction

In his influential early paper on data types[4], Hoare states:

Every value belongs to one and only one type.

It is evident from Algol-style languages like Ada and Modula that many programming language designs conform with this view.

The adoption of this view bought compiler simplicity at the expense of language expressivity — a reasonable trade off then, but not now. In recent languages, such as Oberon, some of the lost expressivity is regained by the introduction of subtypes[13].

It is shown in section 10 that, in real life, types intersect in ways that are difficult to model using only subtyping. A closer fit between reality and model can be achieved if types are allowed to intersect in arbitrary ways. The aim of the paper is to examine some of the consequences and to suggest how arbitrary intersections can be constructed.

2 Types as sets

If one views a type as merely a set of values, type intersection is straightforward to define and understand (it is merely set intersection). If one adopts the classical view that a type is an algebra[8], however, it is much more difficult to define the meaning of intersection.

The author contends that it is not essential to view a type as an algebra. Types are useful because they provide structure and allow type checking. This can be done even if one views types merely as sets of values. In the rest of this paper, therefore, types are sets of values and $\mathcal{U}$ will denote the set of all the values that can belong to a type.

The author contends further that in programming languages types play only one role: they serve as constraints on the domains and codomains of functions. That is, if $\mathcal{D}$ and $\mathcal{C}$ are types, and $f : \mathcal{U} \rightarrow \mathcal{U}$ is a function with domain constraint $\mathcal{D}$ and codomain constraint $\mathcal{C}$, then all programming languages require the definition of $f$ to conform to the following:

\[
\begin{align*}
    f(x) &= \text{Undefined}, & \text{if } x \notin \mathcal{D} \\
    f(x) &\in \mathcal{C} \cup \{\text{Undefined}\} & \text{if } x \in \mathcal{D}
\end{align*}
\]

Typically, an error is signaled when $f(x)$ returns Undefined (not be confused with non-termination or Bottom). In statically typed languages such errors are sometimes predictable by the compiler.

As a syntactic convenience, we will write $f : \mathcal{D} \rightarrow \mathcal{C}$ to show that $f : \mathcal{U} \rightarrow \mathcal{U}$ is a function with domain constraint $\mathcal{D}$ and codomain constraint $\mathcal{C}$.

The reader may be forgiven for being sceptical about the contention that types only serve to constrain functions. After all, types are used to declare variables and check assignments. What's that got to do with functions?

Everything, as long as we regard variables as functions whose definitions are provided and updated by assignment statements. In other words, a variable declaration such as
V : Some-type
must be regarded as syntactic sugar for

\( v : \{ \} \rightarrow \text{Some-type} \) is a function whose
evaluation rule will be supplied later on
by an assignment statement and may
be changed (as time progresses) by fur-
ther assignment statements.

We will furthermore regard V to stand for \( v() \)
when used in an expression (\( v \) can be denoted
as Address(V)).

3 Related work

Value sharing (type intersection) has received
scant attention from researchers working on type
systems. For example, Cardelli[2] mentions it in
passing as a special case of parametric polymor-
phism but gives no explicit consideration to it.
Language designers either dispense altogether
with the notion of typed variables (for example
Icon[5] and SETL[9]) or make every effort to
restrict value sharing.

As far as the author knows, the work of Bailes
on G[1] has been the sole exception to this
trend. Bailes, however, arrives at value sharing
as a necessary consequence of his well-motivated
overall approach and gives little specific atten-
tion to it.

4 Type equivalence

The question of when type A is equivalent to
type B has been difficult to resolve using other
views of what a type is. If, however, types are
viewed as sets that merely serve as domain and
codomain constraints on functions, there is only
one sensible answer: type A is equivalent to type
B, if and only if the same set of values make up
both type A and type B.

Thus, if we declare

\[
\begin{align*}
type\ A &= \text{array} \ 1..10 \ \text{of} \ \text{Int} \\
type\ B &= \text{array} \ 0..9 \ \text{of} \ \text{Int}
\end{align*}
\]

we can determine whether types A and B are
equivalent merely by determining whether set A
is equal to set B.

Of course, the difficult question now becomes
the interpretation to be given to type construc-
tors. Most type constructors have straightforward
interpretations, as can be seen from the following examples:

\[
\begin{align*}
\text{set of } T &= \{ S \mid S \subseteq T \} \\
\text{array } R \text{ of } T &= \{ f \mid f : R \rightarrow T \} \\
\text{pointer to } T &= \{ f \mid f : \{ \} \rightarrow T \}
\end{align*}
\]

We thus can readily conclude that types A
and B above are not equivalent since \( f \neq g \) for
all \( f : \{ 1 \ldots 10 \} \rightarrow \text{Int} \) and \( g : \{ 0 \ldots 9 \} \rightarrow \text{Int} \). But what about the following types?

\[
\begin{align*}
type\ C &= \text{record} \ x : \text{Int}; \ y : \text{Char} \ \text{end} \\
type\ D &= \text{record} \ x : \text{Int}; \ y : \text{Char} \ \text{end}
\end{align*}
\]

The question should be familiar to partici-
pants in the name equivalence versus structural
equivalence debate. By interpreting types as
sets, the question is transformed, however, from
"Are types C and D equivalent?" into "How
does one characterize the set of values repre-
sented by a record constructor?". Neither ques-
tion has an answer that will satisfy everyone.

Given our convention of interpreting a vari-
able name, such as R, as syntactic sugar for \( r() \),
it seems reasonable to regard \( R.x \) as syntactic sugar for \( r() (x_C) \) and \( S.x \) as syntactic sugar for
\( s()(x_D) \), if \( R \) is of type C and \( S \) of type D. This
requires the following interpretation of type C
as a set:

\[
\begin{align*}
\{ f \mid f : \{ x_C, y_C \} \rightarrow \text{Int} \cup \text{Char} \\
&\text{and } f(x_C) \in \text{Int} \text{ and } f(y_C) \in \text{Char} \}
\end{align*}
\]

Thus, if \( x_C = x_D \) and \( y_C = y_D \) then types C
d and D are equivalent. We now have two sensible
choices:

1. \( x_C = x_D = 'x' \) and \( y_C = y_D = 'y' \).
2. \( x_C, y_C, x_D \) and \( y_D \) are distinct values.

Selecting the first choice amounts to a form of
structural equivalence that allows the conclusion
that type E, below, is also equivalent to types C
d and D.

\[
type\ E = \text{record} \ y : \text{Char}; \ x : \text{Int} \ \text{end}
\]

Selecting the second choice amounts to name
equivalence.

5 Assignment compatibility

By our earlier definition, a variable is a function
and the value returned by this function must be
a member of the set \( T \cup \{ \text{Undefined} \} \) where \( T \)
is the type of the variable. When an expression
is assigned to a variable, it is thus necessary to
ensure that the expression results in a member
of \( T \cup \{\text{Undefined}\} \) before allowing the assignment to take effect.\(^1\)

Since operators are also functions, expressions can be regarded as function applications. The type of an expression is thus the range set of the (outer) function that is being applied. This is guaranteed to be a subset of \( C \cup \{\text{Undefined}\} \) where \( C \) is the codomain of the function.

Now, if \( C \subseteq T \) or \( C \cap T = \{\} \), the validity of the assignment can be decided without evaluating the expression. Thus, a language designer can ensure that all type checks can be done by a compiler, simply by insisting that assignments are illegal unless \( C \subseteq T \).

Compile-time (or static) type checking is generally regarded as highly desirable and hence many language designers claim that their languages allow static type checking. In practice, however, static type checking is so restrictive that no realistic language enforces it. Virtually all languages allow assignments to be carried out if \( C \supseteq T \) and generate run-time checks to ensure that the assignments are legal. The main difference from language to language is whether the language forces the programmer to make the type check explicit or not.

If types are to intersect freely, however, it is not reasonable to insist that \( C \supseteq T \). Assignments should be allowed, subject to run-time checks if necessary, whenever \( C \cap T \neq \{\} \).\(^2\)

Of course, this means that many type errors that are easily detected by the compilers of languages like Ada and Oberon, become very difficult for a compiler to detect, thus requiring the generation of expensive run-time checks. The author contends, however, that it is no longer appropriate to restrict the expressivity of a language merely to limit the complexity of compilers and run-time systems. Such a strategy does not get rid of the complexity, it merely dumps it over the fence in the application programmer’s yard. Furthermore, much progress is being made on the optimization of “dynamic” languages (for example [3]) and it seems reasonable to expect that such compilers will convert most run-time errors into compile-time errors.

That is, while static languages may be safer than dynamic languages at present, they are not inherently safer.

\(^1\)Usually, assigning Undefined to a variable will also result in a run-time error.

\(^2\)This is already the case for subrange types in most languages. The argument here is that it should apply to other types as well.

6 Problems with records

Although the second interpretation for record constructors (amounting to name-equivalence) has been dubbed sensible in section 4, it does not seem sensible to adopt this convention in a language where types are regarded merely as sets, since the obvious interpretations of the sets of values generated by the other type constructors all amount to structural equivalence.

Structural equivalence, however, creates problems when records are used as the “concrete” representations of abstract types. For example, if

```plaintext
[Type Stack =
  record
  Top : range 0..100;
  Elems : array 1..100 of Int;
  end record]
```

and

```plaintext
[Type Funny-rec =
  record
  Top : range 0..100;
  Elems : array 1..100 of Int;
  end record]
```

then types Stack and Funny-rec represent the same set of values if structural equivalence is used.

Given assignment compatibility as outlined in the previous section, this means that a client programmer can always make the “hidden” representation of abstract types visible by providing alternative type definitions that happen to describe the same sets of values as the abstract types. Thus, abstract types have to be handled in non-traditional ways if types are to intersect freely.

7 Types as values

One intuitively expects to be able to declare variables of type “set of something”. If one allows such types, it implies that one allows sets to be elements of sets, which implies that one ought to allow types to be values.\(^3\)

Of course, if a programming language allows very large sets, like Int, to be regarded as values,

\(^3\)If \( \mathcal{U} \) is treated as a value, it becomes possible to specify paradoxical sets, for example \( \{ S \subseteq \mathcal{U} | S \notin S \} \).

To avoid paradoxes we decree that \( \mathcal{U} \in \mathcal{U} \) and \( \mathcal{U} \subset \mathcal{U} \) by definition and that all paradoxical set formers must result in Undefined.
the language implementations must resort to a form of lazy evaluation: one would not wish the compiler to generate code that fully evaluates a set like \( \{ x \in \text{Int} \mid \text{odd}(x) \} \) if the set is only used as the target of a members test.

8 Dynamic Types

Most programming languages require all the values making up a type to be determinable during compilation, since this is a prerequisite for static type checking. If one drops the insistence that type checks should be static where possible and very simple otherwise, it becomes possible to consider whether sets of values, that vary at run-time, should be allowed to serve as types.

Such “dynamic” types would be very useful in database-oriented languages. For example, one would be able to declare:

\[
\begin{align*}
\text{Parts} & : \text{set of } U \\
\text{Suppliers} & : \text{set of } U \\
\text{Parts-supplied} & : \text{Suppliers} \rightarrow \text{set of Parts}
\end{align*}
\]

Since Suppliers is a variable, values of type Suppliers can be created and destroyed as time progresses, simply by including and excluding values from set Suppliers. Moreover, the domain constraint of function Parts-supplied dynamically adjusts to the creation and destruction of supplier values. (In other words, the type system takes care of referential integrity.)

Although any sets of values can make up types Parts and Suppliers, it is better not to press numeric, string, array or record values into service as part or supplier values. Parts and Suppliers are abstract data types and a means must be provided to populate them with abstract values. In other words, it must be made possible to get hold of values that belong to \( U \), but to no other set. Providing a built-in function called new comes to mind.

By making it possible to create new values at run-time and to include these values into one or more run-time sets that can serve as types, one makes it possible for types to intersect in arbitrary ways. With classical sub typing, the set of types to which a newly created value can belong can be determined by the compiler and types can only intersect if the one is a subtype of the other or if both types share a common subtype.

9 Object Classes

Dynamic typing subsumes sub typing and provide for the integration of object-oriented concepts and Algol-style languages in a natural way. For example

\[
\text{Suppliers} : \text{set of } U
\]

can be seen as the declaration of an object class and

\[
\text{Parts-supplied} : \text{Suppliers} \rightarrow \text{set of Parts}
\]

can be seen as the declaration of a member variable for class Suppliers. Dynamically creating new abstract values and including them in set Suppliers amounts to dynamically creating objects of class Suppliers.

Derived classes correspond to subtypes, and inheritance follows naturally. For example, given type (class) \( A \), an operator (method) \( f : A \times A \rightarrow A \), and a subtype (derived class) \( B \subseteq A \), then \( f \) may obviously be applied to arguments of type \( B \). Moreover, the value resulting from the application of \( f \) to arguments of type \( B \) may well be a member of \( B \) and thus may be assigned to a variable of type \( B \), subject to a “run-time” type check. In other words, subtype (derived class) \( B \) effectively inherits operator (method) \( f \) from supertype (super class) \( A \).

Inherited methods can be overridden by allowing operators (methods) to be overloaded. However, operator/function names cannot be as freely overloaded as in a language like Ada. For example, if

\[
f : A \rightarrow C, \ f : B \rightarrow C, \text{ and } A \cap B = \{ab\}
\]

then it is impossible to determine which \( f \) must be used when one encounters \( f(ab) \).

Similar difficulties arise in object-oriented languages that allow multiple inheritance and similar solutions can be adopted.

10 Beyond Classes

Object-oriented programming languages are currently advocated by many as the long awaited solution to the software crises. For the most part, the author agrees that these high expectations are justified. However, it should not be imagined that these languages have advanced programming language expressivity to the natural limits.
For example, consider an object-oriented design for a university database. It seems likely that the designer will have introduced a class Person with sub classes Student and Staff-member. The usual interpretation of classes as types require Student and Staff-member to be disjoint. In real life, however, these sets are not disjoint and the designer thus will have to go further.

In simple cases, multiple inheritance seems the answer. If the designer adds a Student-staff-member class, which inherits from both Student and Staff-member, to the database schema the problem seems solved: An instance of Student-staff-member ISA Student and ISA Staff-member, the latter two classes are thus no longer disjoint.

Consider, however, the complications that arise when using this technique to model the real life intersections between classes Student, Staff-member, Parent and Sports-club-member. Must the database designer anticipate the need for the class Sporting-parent-student-staff-member? Surely not!

Consider further, the complications that arise if the designer decides to introduce sub classes Academic-staff-member, Administrative-staff-member, Technical-staff-member, and so on. In this case, creating an instance of Student-staff-member creates a staff member object that belongs to none of the sub classes of Staff-member. The designer thus is forced to remove class Student-staff-member from the schema and replace it with Student-academic-staff-member, Student-administrative-staff-member, and so on.

The author suspects that the database designer will simply give up and provide a schema that is an oversimplification of the real world. The application programmer ends up having to solve the problem, typically by adding explicit run-time type checks all over his code.

If the designer can use dynamic types that can intersect in arbitrary ways, the above modeling problems simply do not arise.

11 Summary

The consequences of completely abandoning the view that every value belongs to one and only one type are profound. One can achieve no more than a partial abandonment, however, unless one also holds that a type is simply a set of values and allow any set of values to serve in the role of a type.

The interpretation of a type as no more than a set of values forces one to conclude that type constructors are simply set operators. This in turn simplifies the resolution of questions about type equivalence and suggests that structural equivalence is appropriate.

Structural equivalence leads to problems when applied to record constructors. Records have also proved troublesome to database designers[6] and have been successfully done away with in DAPLEX[10]. This suggests that records are not "nice" constructs and that "post-Hoare" programming languages might better off without them.

Another consequence of regarding types as nothing more than sets of values, is that operators take on an independent existence and that the notion of operator inheritance by subtypes thus becomes simple. Also, types can be made into first-class values by treating sets as first class values. This is more intuitive than other approaches[7], but requires implementers to tackle lazy evaluation. Finally, polymorphism and the ability to do type membership tests are automatic consequences of making types equivalent to sets.

In classical Algol-style languages the prevalence of disjoint types and highly restricted mechanisms for introducing subtypes make "nearly" static type checking possible. When any set of values can serve as a type, however, it must be accepted that non-trivial run-time type checks will have to be performed in some situations — typically those situations where the application programmer would have to provide them if the language does not.

Acceptance of non-trivial run-time type checking makes it reasonable to accept furthermore that, in some cases, even the values making up a type will not be known at compile-time. This leads to the very useful concept of a dynamic type that makes it possible to support abstract types in a non-traditional, more natural way, and to integrate concepts from database languages and object-oriented languages into Algol-style languages.

The author has recently designed an Algol-style language that supports and exploits unrestricted type intersection. A prototype version has been in use since early 1989 and our experiences with its expressivity are very positive. The design of the type system of the language is re-
ported in more detail in [11] and the language itself is defined (in under 20 pages) in [12].

The author has little doubt that the active exploration of the design space of languages that allow and exploit unrestricted type intersection will result in some very exciting and powerful new programming languages appearing on the scene.

12 Conclusion

If one assumes that the "software crisis" will not allow restrictions on the expressivity of programming languages to survive advances in compiler technology, it is also safe to assume that statically typed languages like Ada, C++, Eiffel and so on have no future.

The author believes that the future belongs to persistent, polymorphic, dynamically typed, object-oriented languages that allow types to intersect freely. This article has shown that these need not be radical languages that are as different from Algol family as LISP, Prolog or Smalltalk are, but can just as well be Algol-style languages.

References


A TRANSPUTER - BASED EMBEDDED CONTROLLER DEVELOPMENT SYSTEM

M R Webster, R G Harley, D C Levy, D R Woodward
Electrical Engineering, University of Natal,
King George Vth Ave, Durban, 4001

ABSTRACT
This paper describes a software development environment currently being built at the University of Natal that provides a powerful and easy to use engineering tool for the design, simulation and implementation of high performance embedded controllers using parallel processing suitable for high speed motor control through to robotic applications. The development of a scalable controller platform based on the transputer is discussed in detail. This paper presents the functional specifications and discusses some of the software and hardware related issues/features to enable the system to achieve the required high performance, hard real-time operation.

1. INTRODUCTION.
An electrical motion control system may be represented by the block diagram of figure 1 [1]. Due to advances in power-electronics and micro-electronics technology during the past decade, modern motion control systems are becoming increasingly more intelligent and sophisticated [2], and features such as self-tuning, self-commissioning, fault diagnostics, fault tolerance, protection and remote communication are now possible. This allows relatively unskilled labour to locate and repair faults and avoid unnecessarily long service interruptions, and also to reduce time and costs during the development and commissioning of systems.

Modern motion control systems are multi-disciplinary in nature [2,3,4], hence their design and development is both labour and time intensive and therefore expensive. Furthermore, the demand for improved features and higher performance calls for more sophisticated and complex control and therefore more processing capability and throughput from the digital controller; thus motion control technology is becoming very knowledge intensive and requires a substantial and well coordinated team effort to produce a robust yet flexible and feature filled motion control system at market related prices. This situation is being further exacerbated by the growing demand for automated systems which for example might require several multi-axis robotic systems which together constitute a real-time high performance distributed control system.

It was for these reasons that the design of a software development environment for high performance, scalable, multiprocessor-based controller was undertaken. Although the system is suitable for many hard real-time embedded control applications, its first application will be to motion control, and it will be referred to throughout this paper as the Motion Control Development System (MCDS).

2. SYSTEM OVERVIEW.
An overview of the Motion Control Development System (MCDS) is shown in figure 2. The system consists of a PC host connected to the Programmable High Performance Controller (PHPC) via a communications link. The PC hosts the design and visualisation environment which can be removed.
once the control software is embedded in an application. The PHPC contains a number of transputer-based cards which provide processing and I/O capability to form a scalable, digital controller platform. The PHPC passes control signals to the actuators in the plant (an inverter in the case of a motion control application) and interprets feedback signals from sensors measuring plant variables (current, voltage, temperature, speed, position, etc). To develop and implement real-time digital control strategies a number of software modules are provided as shown in figure 2. The overall structure of the software for the development system is shown in figure 3. The sections that follow provide an overview of the various sub-systems that constitute the MCDS. Each of these sub-systems are covered in more detail in [13 - 18].
3. THE GRAPHICAL USER INTERFACE.

The Graphical User Interface (GUI) [5] provides a programming and visualisation tool for the MCDS. While graphical representation of computer programs is a controversial topic and has produced disappointing results for general purpose programming, it has been successfully applied in signal processing applications [19] and we believe it is suitable for the design of embedded hard real-time software for motion control. The user interface can be divided into two main areas: the development environment, and the on-line visualisation and interaction environment, as shown in figure 4. The development environment enables the user to create, on a Personal Computer (PC), in a windows-based environment, a block diagram representation of the desired control strategy, using a Function Block Editor (FBE). The FBE generates a netlist (a matrix describing an interconnection network) of the block diagram description of the control algorithm. The netlist forms the input to the CASE tools, which generates the necessary code as specified by the block diagram to form the actual digital control strategy required. The on-line visualisation and interaction environment passes data gathered by the transputer network of the PHPC to the PC for graphical display. The user is able to change parameters on-line from the PC; these are then relayed to the transputer network. A detailed functional diagram of the GUI is shown in figure 5.

- **The Function Block Editor [FBE].**
  The FBE allows the user to design a control system in the form of a block diagram in a windows-based environment. The blocks are icon representations of the actual control algorithms. The FBE records how the blocks are connected, together with the specified visualisation points and sampling rates. This constitutes the important netlist information generated by the FBE.

- **The Icon Editor [IE].**
  For generality, the user is able to design or customize icons in association with writing a corresponding new control algorithm. The interactive graphics editor allows the user to allocate or associate a control algorithm with the customized icon.

- **The icon library.**
  Once the icons are created in the icon editor they can be stored in an icon library for later use. However, not every user will want or need to create customized icons. Standard library icons required in a typical real-time environment will be provided, and will form a starting point for each user.

- **The source code library.**
  Associated with each icon is a module of code. This code is an implementation of the mathematical equations used to describe the function (i.e. integrator, summer, multiplier, filter, etc.). Each block of code associated with a specific icon has inputs and outputs. These I/O points in the block of code receive/transmit data from/to other blocks of code associated with other icons. The block diagram netlist generated by the FBE is used to determine how the blocks of code are interconnected. The user may create his own code modules, which are checked by a CASE tool called the Code Parametiser (CP), which is described later, to determine if they conform to the format understood by the MCDS.
The on-line visualisation and interaction environment.

Once the controller is in a run-time mode, on-line data capturing, visualisation and analysis may be performed. The supervisor process collects data from the transputer network (sampled by the RTK) and passes data to the PC's graphics handler at the user's request. The GUI running on the PC would have switched from a development mode to an on-line visualisation and interaction environment. The user is thus able to interact with the control algorithms, tuning them on-line. This feature is also important for fault diagnostics.

4. THE CASE TOOLS

In order to automate the generation of robust real-time code and to ensure efficient parallel execution the following CASE tools are being developed:

4.1 The Serial/Parallel Load Analysis Tool (SPLAT).

Due to the scalable nature of the PHPC and the different applications to which it will be applied (e.g. single and multiple AC drive systems), a means of optimally matching the software tasks specified by the control strategy and the hardware resources available is required. This function is provided by the Serial/Parallel Load Analysis Tool (SPLAT), see figure 3. SPLAT obtains information about the structure of the controller strategy from the output of the Function Block Editor in the Graphical User Interface, and details of the transputer network of the PHPC or multiple PHPCs from the Network Analyser. Using this information as well as information which characterises the control algorithm code represented by the icons of the GUI, SPLAT performs the function of a static load balancer to effectively distribute the work load amongst the PHPC processing resources, to meet the performance requirements. SPLAT thus allows the controller to be optimised for either economics i.e. using a single CPU, or performance via parallel processing on a moderate number of CPUs.

4.1.1 SPLAT inputs.

The SPLAT requires the following input information:

- **From the graphical interface.**
  A netlist of the function block specification which together form the complete control system. This will include a time constraint matrix which characterises the performance requirements of each section or loop of the control algorithm by specifying the loop sampling rates.

- **From the function block (control task) library.**
  Initially this will comprise of pre-written functions in Occam or transputer assembler (GUY) code. Other languages such as fortran, C or ADA can be supported as foreign code inserts in an Occam harness. Each function module will be characterised and contain information regarding parameters such as execution time, memory and I/O requirements and a ID tag. These modules would also define the granularity of parallelization possible in the system as they will form the smallest units of computation.

- **From the network analyser.**
  A situation report of the number, types and status of all the cards in the PHPC rack, as well as the processor and I/O interconnection topology.

4.1.2 SPLAT processing.

From the aforementioned stage, the SPLAT will generate a mesh representing the absolute timing constraints imposed by the harness (RTK), in which the required function blocks of code will be filled in. Using optimization techniques, the function modules will be placed on each processor and linked to each other. SPLAT actually manipulates a Computer Activity Flow Graph (CAFG). The SPLAT uses graph theory and Critical Path Analysis to determine heuristically the optimal code placements to satisfy the real-time criteria.

4.2 The Source Code Linker (SCL).

The Source Code Linker is required to take the output of the SPLAT (a static schedule) and retrieve the relevant blocks of source code from the function block library. The SCL then links these modules into the Real-time Kernel to form a complete source code program for the specified multiprocessor network. The SCL is also required to link the drivers required by each I/O module used in the system. This allows the assignment of I/O addresses to be automated. The complete parallel source code is then passed to the Occam compiler and network loader routines for compilation and distribution to the controller network.
4.3 The Code Parameteriser (CP).
The user may wish to generate custom control or signal processing functions for algorithms not provided for in the standard library. The Code Parameteriser is required to extract information about the user-defined modules so that these modules may be manipulated by the SPLAT. Typically the worst-case execution time, the memory requirements and I/O connections are needed. The CP also checks the syntax of the user-defined code.

5. THE REAL-TIME KERNEL.
The Real-time Kernel (RTK) is the collective term for those program modules whose function it is to manage all hardware and software resources within the real-time environment [6]. The real-time harness includes features such as task scheduling, communication harness, sampling, fault-tolerance, interrupt handling and configuration. The main specification for the proposed controller is that it is capable of 20 kHz sampling rates with less than 10 μs interrupt latency. Besides this, the controller must also be configured automatically and be capable of implementing various parallelization strategies such as farming and data flow. If transputers are to be used in high performance control applications, it is essential that the I/O and event handling capabilities of the transputer are well understood. This required insight into the hardware architecture and the low level language (GUY code) of the transputer [8,9]. The interface between the transputer executing the corresponding control process, and the external event generating devices can be divided into two main processes, namely the event handling and the data handling. For maximum performance, these processes must be handled as quickly and efficiently as possible. Real-time controllers are often connected to a wide variety of devices for monitoring sensors and controlling actuators. These devices convey and retrieve information to and from the controller via I/O registers of some form. Events may be generated in order to signal to the controller that certain operations must be performed or that an error condition has arisen. For loop sampling frequencies of up to 20 kHz, the controller only has 50 μs to retrieve, process and output data. The processing component of the system, which may involve complex control algorithms, can be implemented with parallel processing, thereby substantially reducing the controller delay. Therefore the problem is to reduce the I/O delay to a minimum. This is achieved by using an interrupt driven process as the only high priority process on the transputer, triggered by a signal on the Event Req pin, and using memory mapped I/O to read the I/O registers; times of 8.5 μs have been recorded from the time the event occurred to the time the data was sent down a hard link, therefore leaving about 40 μs for signal processing [7]. The RTK also extends the priority levels available on the transputer to an arbitrary number, as well as providing a pre-emptive scheduling strategy as opposed to the existing microcoded round-robin scheduler. The RTK scheduler is transparent to the rest of the controller tasks on the transputer which can take the form of a normal Occam task force.

6. THE CONTROLLER HARDWARE.
Two compatible but distinct controllers are under development. These are:
- The Embedded Programmable High Speed Controller (EPHSC) [12],
- The Programmable High Performance Controller (PHPC).

The EPHSC is based upon a dual-port memory concept and is designed for small embedded applications. The EPHSC concept is shown in figure 6. The inclusion of the dual-port allows numerous devices of differing speed to be connected onto the transputer bus. Essentially this controller uses a mailbox concept. The specifications for the prototype are included in the appendix. The EPHSC has been designed to accommodate 8 dual-port I/O modules.

The PHPC is implemented on a cell concept, each card in the system
contains a transputer cell. The cards either perform a processing or I/O function. Interconnection is achieved by a configurable network of transputer serial links. Each I/O card consists of a transputer cell plus an I/O sub-system, as shown in figure 7. The I/O sub-systems include digital to analog converters, analog to digital, digital to digital, and many others. The transputer cells are identical, and the I/O sub-systems merely plug onto the transputer cells to give them their personalities. The EPHSC described above forms a complex I/O module for the PHPC. The transputer cells and I/O sub-systems form cards which are grouped together in a rack. Each rack can house up to 8 transputer cards and one crossbar card. Racks are connected together by means of external link connections, which can be optic fibre based for long distances. The backplane of the rack consists of power, control and link buses. The power bus has been designed specifically for a mixed analog-digital system, and provides separate power lines for the digital and analog cards. Also included is a zero current ground reference, allowing for accurate analog signal measurement. In many existing systems insufficient attention has been paid to noise pickup in analog signal lines, making them inaccurate in noisy industrial environments. The full control bus is accessed by all cards and is used to carry triggering/synchronisation information. It is absolutely necessary in a real-time control system to ensure certain events occur simultaneously, and this has traditionally been difficult to achieve in a distributed network of processors, where it can only be guaranteed that a certain process will run within a given time period. By providing a separate control bus, processes can be forced into synchronisation with critical events. The control bus accesses sub-system hardware directly, and hence time-critical functions are made independent of transputer interrupt latency. For example, a control system typically needs to sample an analog signal at a regular and precise interval. It also has to update the processed digital output as an analog value at the same instant or time tick. The control bus supports extensive event triggering capabilities. Up to 8 asynchronous triggers can occur simultaneously on the bus. Triggering can be sourced from individual controller cards, a programmable timer card or an external source. Multiple grouped triggering is also important in a controller, and this is achieved by mapping the relationship between control lines and the action taken by a card. For example, to measure the instantaneous power in a load, a voltage and a current measurement must be made. If these are not sampled at the same instant, this will cause errors in estimating the instantaneous power, which will compromise the control strategy.

A large degree of software configurability is incorporated in the controller. This allows rapid system re-configuration. Link routing between transputers is configurable from the master transputer and is performed by a crossbar switch card. This card can range in complexity from a simple non-configurable jumper board up to a multiple electronic crossbar switch arrangement. The I/O sub-systems have a large
degree of testability built in, for example, the A/D subsystem has the capability to calibrate, on-line, its inputs from a switchable reference. This allows the controller to check its operational integrity while the system is running.

7. CONCLUSIONS.
This paper has described the implementation of high performance, transputer-based Motion Control Development System (MCDS). Preliminary results have been reported in [10,11,12]. The primary function of the MCDS is to provide a scalable controller platform capable of solving small single AC drive motion control applications using an economical, single CPU controller through to complex multi-drive motion control applications requiring multiprocessor performance. It is also envisioned that the controller will be suitable for a large number of other applications that are computationally demanding such as turbine control, robotics and flight control systems.

8. REFERENCES.
9. APPENDIX

9.1 PRELIMINARY HARDWARE SPECIFICATIONS

9.1.1 Embedded Programmable High Speed Controller (EPHSC).

Motherboard
Processor: T800 25 MHz (or compatibles T425, T801)
Memory: 2 Mbytes DRAM
256 Kbytes EPROM
32 Kbytes Non Volatile RAM
I/O Bus: 16 bit memory mapped
Trigger channels: 2
Communications: 4 x 20Mbit/s INMOS serial links

Daughter Boards
1. Analog to Digital Board
No. of Channels: 4 channels with sample and hold.
Sample Modes: Simultaneous or independent
Accuracy: 12 bit resolution
Conversion time: 5 µs maximum
Voltage Range: +/- 10 Volts
Input Impedance: greater than 10 MOhm

2. Pulse Width Modulation Board
No. of phases: 3
Modulation Method: Space Vector
Switching frequency: up to 18 kHz
Compensation: Dead-band and commutation delay time
Vector description: Polar or cartesian

3. Shaft Encoder Board
Resolution: 16 bits
Shaft encoder type: incremental
Input Signal type: Two quadrature square waves plus index signal

4. Digital Board
No. of channels: 2 input or output (programmable)
Resolution: 16 bits per channel
Voltage Level: TTL

9.1.2 Programmable High Performance Controller (PHPC).

Transputer Cell
Processor: This consists of a single transputer and must be either a T414, T425, T800 or T801.
Memory: 8-bit SIMM modules of 256 Kbytes or 1 Mbyte gives 1, 2, 4 or 8 Mbyte. 60, 80 or 100ns
I/O Bus: 16-bit high-speed memory mapped interface to subsystems
System Connection: Buffered TRAM standard interface to backplane
Description: Any transputer compatible with the T800 may be used in the card. Maximum clock speed is 25 MHz.
The card backplane carries TRAM standard signals. Control lines from the card may also be connected in parallel in order to improve reliability over the daisy-chained TRAM format. High speed dynamic RAM modules, each of 255 Kbytes or 1 Mbyte capacity, give the cell from 1 to 8 Mbytes of memory operating with an access time of 60 ns and cycle time of 200 ns. The memory configuration may be optimised for the type and speed of memory used. No parity checking is to be incorporated in the prototype. Each Transputer Cell in the system has its own electronic serial number. This simplifies the autoconfiguration software. TRAM standard subsystem ports are incorporated as well as a port for identifying the master transputer slot.

Analog to Digital Converter
No. of Channels: 4 analog input channels.
Accuracy: 12-bit resolution per channel, accurate to 11-bits.
<table>
<thead>
<tr>
<th>Sampling Rate:</th>
<th>Minimum conversion time of 3 μs, all channels triggered simultaneously.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voltage Range:</td>
<td>+/- 10V inputs, protected to +/- 25V</td>
</tr>
<tr>
<td>Input Impedance:</td>
<td>greater than 10 MΩhm</td>
</tr>
<tr>
<td>Other Functions:</td>
<td>4 channel clocks, 3 external interrupts, Subsystem ID, High-speed I/O structure, Testability built in, Built-in precision reference, Offset and Span testing, Special programmable sampling modes viz. grouped simultaneous and multirate sampling, Overrun detection.</td>
</tr>
</tbody>
</table>

**Description:**
The analog to digital board plugs into a transputer cell to give a high performance A/D system. Each analog channel features its own sample & hold and high-speed 12-bit converter. The converters are available in speeds of 3 to 12 μs. The high-speed sample and hold can be used to undersample at up to 6 MHz. Each analog channel has a +/- 10V input range which is protected up to 25V. Each channel may be electronically switched to a 0V or 10V internal reference source in order to check the offset and span of a channel. Channels are triggered by channel clocks which are connected to the control bus. The mapping between the control bus and a channel is software programmable and allows for true and complex grouped simultaneous sampling to be performed. Individual channels and a group of channels to be simultaneously sampled may be triggered from software. Hardware checks for overrun (which indicates that samples are being requested faster than the overall controller can process them). Three additional external interrupts have been added to extend the transputers' single interrupt line. The entire card is testable from software, allowing the operation of every single I.C. to be verified.

**Digital to Digital Card**
- **No. of channels:** 8 8-bit I/O ports, Ports may be programmed for 8 or 16-bit operation
- **Voltage Range:** CMOS compatible
- **Output Circuitry:** Choice of tristate or open-collector I/O ports, Transparent or latched inputs
- **Other Functions:** Full hardware controlled handshaking for peripherals, Subsystem ID.
- **Description:** This card must be connected to a transputer cell. There are 4 groups of 16-bit ports. These may be programmed as fully independent 8-bit ports. Each port has three control lines; read, write and acknowledge. These allow the ports to appear as dual-port memory locations to intelligent peripherals but also enables the transputer to be able to drive dumb devices and read in switch lines.

**Digital to Analog Card**
- **No. of channels:** 4 12-bit outputs, 4 8-bit outputs.
- **Other Functions:** All outputs are dual-rank, Grouped and complex mapped output updating, Output overrun detection, Precision reference on board, Built-in testability, Analog to digital loopback to enable testing of channel outputs, Subsystem ID.
- **Description:** This card must be connected to a transputer cell. Channel outputs are updated by signals from the control bus. The mapping between control signals and updating is software programmable. It is possible to update an output signal at the same instant as an A/D channel is sampled by the A/D board simply by programming in the correct mapping information. In this way the entire control system may be synchronised to an external timing tick source. This card is virtually a reverse implementation of the A/D card.

**Crossbar Card**
- **Processor:** Either T414, T425, T800 or T801
- **No. of links:** Full 32x32 transputer link multiplexer
- **Description:** This card is software programmable from the Master transputer card to electronically configure the multi-transputer link communication structure. The crossbar routes the serial communication data between card slots and also interfaces to other racks. The crossbar is made up from a COO4 or compatible chip.

**Processor Cards**
- **Processor:** Two sizes are available; a 4 transputer cell and an 8 transputer cell, populated with either T414, T425, T800 or T801 processors.
- **Description:** These cards are used as computational engines. Each transputer has up to 4Kbytes of internal RAM and provisionally 2 Mbytes of external RAM. It is intended that their communication links are electronically configured via the crossbar switch.

**Timer Card**
- **Processor:** Either T414, T425, T800 or T801
- **No. of Clocks:** 8
- **Max. Frequency:** 20 MHz
- **Min. pulse width:** 50 ns
- **Description:** This card contains software programmable, external timers which will enable the system clocks and event generation of the back plane control bus. It is a generic programmable pattern generator and is suitable to generate software calculated PWM signals as well.

**Fibre Optic Link Communications Card**
- **Processor:** Either T414, T425, T800 or T801
- **No. of Links:** 2
- **Description:** This card is constructed from the standard transputer I/O cell and contains a gate array chip that converts the normal transputer links into fibre optic links. This chip provides error checking and facilities for remote resetting and rebooting. The purpose of this card is to provide secure, high bandwidth communications for distributed systems in potentially noisy environments.
Improving A Control and Sequencing Language

Coin A. Fair* and Gabriël D Smit
Laboratory for Advanced Computing
Department of Computer Science
University of Cape Town

May 1991

Abstract

In a process control environment, batch processes, as opposed to continuous processes, are characterised by multi-product manufacturing lines producing relatively small quantities which often involves frequent product changes. One component of batch control systems is a programming language which is used to control and synchronise the operations of the plant. Initially low-level languages (e.g. ladder logic, boolean algebra and assembly language) were used, but have now been replaced by specialised high-level languages. These languages provide more functionality and are easier to use. This paper examines one such a high-level sequencing language (CASL) and identifies functionality, clarity and readability improvements that can be made to the existing language. An implementation of an upwardly compatible compiler for the improved language is described briefly.

1 Introduction

Batch processing provides manufacturers with the flexibility of producing many different products using the same basic equipment, compared to continuous processes which are used whenever large quantities of a product are required and the specification remains relatively constant [8]. The recipes (or sequences) for producing these batch products need to be updated continually in order to maximise the use of raw materials and plant equipment, thus producing cost-effective products. Batch process control has presented unique challenges to the process control fraternity [7, 8, 4, 10] compared to continuous control which is generally well understood and easier to implement [8].

Many low-level and unsophisticated systems have evolved over the years that allow the computer to automatically control a factory or industrial plant [6] and [4]. These systems were often difficult to understand and implement and the need arose for a high level language with concepts and constructs that could map onto typical plant operations. CASL (Control And Sequencing Language) [3], is such a special-purpose language that is used for programming the various functions associated with batch process control. CASL forms part of the CYGNUS process management and control system [2] and was developed in the seventies by ICI (Ltd) in England.

*Partially supported by the Computer Science Development Programme of the FRD
1Pronounced, "castle" or "cassel"
2Registered trade mark of AECI Limited
On current implementations, CASL source code is compiled into an intermediate language. The intermediate code is loaded into a code partition in memory from where it is executed by a run-time interpreter. Loading and replacement of code sequences are allowed during run-time. The interpreter interacts with the plant equipment by reading and/or setting attributes of plant equipment as represented in a real-time database. A separate task, the scanner is responsible for the communication between the plant equipment and the real-time database.

As will be seen, CASL is a somewhat archaic language. The distributor of CYGNUS, AECI Process Computing, approached the Department of Computer Science at the University of Cape Town with a view to developing a local CASL compiler, while at the same time modernizing the language where possible while still maintaining upward compatibility.

In this paper we examine and evaluate CASL and identify weaknesses in its functionality, readability and module clarity; we design improvements that are, as far as possible, upwardly compatible with the old language; and we discuss some issues that arose during implementation of a compiler for the new language using compiler construction tools such as lex and yacc.

2 CASL Overview

This section introduces the concepts and major components of CASL. It is not intended as a language tutorial, but rather to provide the reader with some idea of the nature of the language.

2.1 CASL Operation

In a process control plant several operations (or processes) typically take place at one time. A batch process is defined by Rubin [10] as “having a recognizable start and finish, with one or more states or phases occurring during the operation.” In CASL, these operations are represented by jobs (similar to tasks) which are executed (pseudo-)concurrently. A CASL job can be described as the execution of a sequence. A sequence is similar to a procedure in a conventional high level language and consists of a sequence or block of executable CASL statements. In addition to being called (like a procedure call), a sequence may also be started, in which case a new job is created. A job may consist of a number of sequences because of possibly nested calls to other sequences. However, there is only one active sequence per job (see Figure 1).

Sequences are re-entrant, therefore more than one job can access the same sequence simultaneously. A sequence is subdivided into steps by certain CASL statements. These steps play an important role in the concurrent execution of jobs. While it is the responsibility of the process engineer to subdivide a sequence into steps, in practice these steps occur naturally due to the distribution of the relevant CASL statements in the sequence code.

Pseudo concurrency is obtained by executing part of every job at least once in a given time interval (e.g. once per second). This concept is called a heartbeat. The part of a job that is executed during a single heartbeat is called a phase and corresponds with the execution of a single step in a sequence of code. In other words, one phase of every job is executed every heartbeat. For example, Figure 1 illustrates the conceptual structure of a CASL system being executed in a batch plant. There are n jobs in the plant (CASL
allows a maximum of 60), each with one or more sequences forming part to it. A new job is created when a sequence is started, and this sequence is known as the base sequence. A job may consist of more than one sequence due to nested calls to other sequences, e.g. JOB 3 consists of Sequence 4 (the base sequence) which called Sequence 1, which in turn called Sequence 2. Note that Sequence 2 is accessed (being executed) by JOB 1 and JOB 3. In a single heartbeat n phases will be executed, e.g. in phase 1 step i_1 of Sequence 2 might be executed as part of JOB 1, step j_3 of Sequence 3 as part of JOB 2 in phase 2, step k_2 of Sequence 2 as part of JOB 3 in phase 3, etc.

Phases are not preempted, but run to completion. Mutual exclusion is therefore guaranteed within a step. Synchronisation and scheduling of jobs are obtained with special CASL statements that provide the following operations:

- Start a new job.
- Delay the current job for a given time period.
- Place a given job on hold. The held job could be the current or any other job, and it may be put on hold either immediately or when it reaches a specified point (label) in its sequence.
- Reactivate a given job that has been put on hold.
- Terminate a given job.
The data on which sequence logic operates can be separated into three parts: process database data; PLIST data; and job-local data. By accessing and/or changing attributes associated with plant addresses in the process database, the sequence logic is able to read the plant state and control plant equipment. A PLIST data block is a block of data (or parameter list) that have been grouped together and given a name. A PLIST can be passed as a parameter to a sequence, or can be referenced explicitly as a global data block. A limited number (only four) job-local variables are provided. They have fixed names (X1 to X4) and are local to a job. A reference to such a variable from within any sequence will reference the corresponding variable in the job executing that sequence.

2.2 A CASL Module Definition

A CASL program is usually split into a number of modules. Each module consists of the six sections described next. Some of these sections are optional. An example of a module is given in Figure 2. It has the following components:

TITLE and LET definitions

The TITLE statement defines a module title, while LET definitions define text replacement strings (parameterless macros).

PLIST structure definitions

PLIST structure definitions appear in the PLISTDEF section and are like record definitions in a named type declarations in Pascal and struct definitions in a typedef in C. Every PLIST must have a unique name and contains a list of field definitions. Fields are declared to be of one of five types (or modes in CASL terminology), namely INT (integer), NUM (real), BITS (bit flags), PLAD (plant address), or MESS (message—actually string), and must have a name that is unique within the module. An array is defined by placing the size of the array after the mode keyword. For example

PLIST REACTOR = PLAD OUTLET, INLET, NUM(2) REQVOL, AREA, INT ERRFLAG

defines any REACTOR PLIST to consist of two plant addresses (OUTLET and INLET), two arrays of two real numbers each (REQVOL and AREA), and an integer (ERRFLAG).

PLIST instantiations

The section headed by the keyword PLISTNUM is used to declare the actual PLISTS (PLIST instantiations) for the module. The PLISTNUM section of Figure 2, for example, defines three PLISTS with the REACTOR structure, namely REACTOR(1), REACTOR(4), and REACTOR(9), and one PLIST with the BATCHREC structure. Each instantiated PLIST is created with a list of variables that correspond to the fields defined in the PLIST-structure.

In addition to the instantiation number (the numbers in parentheses), each PLIST must be identified with a unique integer identifier. These identifiers are used by the interpreter as indices to reference the PLIST items in the data partition of the sequence database in memory. Each integer must be unique in a project in order to prevent a conflict from occurring during access. A maximum of 255 PLISTS may be declared in CASL.
TITLE "CASL/TST/18FEB90" EXAMPLE MODULE

LET SCREEN = OCT 100;  %DCP dialogue
LET NL = 10;            %NEWLINE character for printouts

PLISTDEF %Plist definition section
PLIST REACTOR = PLAD OUTLET, INLET, NUM(2) REQVOL, AREA, INT ERRFLAG
PLIST BATCHREC = INT BATCHNUM, SHIFTNUM, MESS SEQTITLE, SEQID

PLISTNUM %Plist declaration(instantiation) section
1 : REACTOR(1)
5 : REACTOR(4)
13 : REACTOR(9)
100: BATCHREC(1)

SEQNUM %Sequence definition section
10: MAINSEQ
20: SUBSEQ REACTOR, BATCHREC
30: ANOTHERSEQ BATCHREC

DATA %Plist initialisation section
REACTOR(4) = '2XV326', '2XV315', 50.0, 60.0, 0.0(2), 22
BATCHREC(1) = 0(2), "Transfer of raw materials for process 10", "X21208"

SEQUENCE MAINSEQ
10:
   IDENT1 "Mainseq"
   DEVIN SCREEN
   DEVOUT SCREEN

20:    X1 = ME    %Set local variable X1 to current job no.
   START SUBSEQ REACTOR(2), BATCHREC(1)
   GOTO 10

ENDSEQ

SEQUENCE SUBSEQ AREACTOR, ABATCH
   IDENT1 "Subseq"
   ABATCH.BATCHNUM = 100
   OPEN AREACTOR.INLET
   ...

ENDSEQ

Figure 2: A sample CASL module.
Sequence identification section

All sequences used in a module, whether they are defined in the module or just referenced in it, must be numbered and listed in the SEQNUM section. Each entry in the section consists of a number followed by a colon, the sequence name, and a list of PLIST-structures that defines the kind of parameter list(s) associated with the sequence. This is similar to procedure prototypes found in other languages. The integer identifiers are used by the interpreter as indices into the code partition of the sequence database. Our sample module identifies three sequences, one (ANOHERSEQ) of which is external to the module. MAINSEQ does not make use of a PLIST, while SUBSEQ requires two PLISTS, namely a REACTOR type PLIST and a BATCHREC type PLIST.

Initialisation of PLISTS

The DATA section contains the optional initialisation of variables in instantiated PLISTS declared in the PLISTNUM section. The PLIST is identified by its name and instantiation number and is followed by a list of data values. The list of values must have a one to one correspondence with the appropriate PLIST-structure definition in the PLISTDEF section.

The list of data values may include a replication factor. This is specified as an integer in parentheses following a data item. For example, in Figure 2, the variables in the PLIST REACTOR(4) are initialised as follows:

```
REACTOR(4).OUTLET = '2XV326
REACTOR(4).INLET  = '2XV315
REACTOR(4).REQVOL(1) = 50.0
REACTOR(4).REQVOL(2) = 60.0
REACTOR(4).AREA(1) = 0.0
REACTOR(4).AREA(2) = 0.0
REACTOR(4).ERRFLAG = 22
```

Sequence definitions

A sequence definition starts with the keyword SEQUENCE followed by the sequence name and a list of formal parameter names. These formal parameters refer to PLISTS of the kind defined in the SEQNUM section. The sequence definition is terminated with the keyword ENDSEQ. CASL labels and executable statements are enclosed between these two keywords.

Executable statements

CASL executable statements can be sub-divided into a number of functional groups. Assignment statements change the contents of PLIST and local variables. Assignments can be made to variables of any type except MESS. However, type conversion is only allowed between integers and reals. Plant control statements allow the manipulation of PLAD’s and their attributes in the database. For example, the value of an attribute may be changed with the SET statement. Program execution control statements direct the flow of control through a sequence. These include an IF THEN ELSE statement, a simple looping construct, a subroutine (sequence) call, as well as a GOTO statement. Text input and output statements read and write text data from and to the current input and output devices.

Job control statements can be used to affect the state and execute positions of not only the current job, but also other jobs. We have mentioned most of these statements under job synchronisation. A further example is the DIVERT statement which is similar to a GOTO, except that it causes another (specified) job to GOTO the given label. Timing and error notification statements form the last two groups of CASL statements.
3 CASL: A Critique

CASL can be evaluated as a batch process control language and as a high level language. In the former, one would ask how well does the language serve the process personnel in specifying control sequences that must be performed in the plant. In other words, how good is the mapping from CASL statements to plant operations? In the second evaluation one would consider such aspects as the clarity and simplicity of the language concepts, clarity of module syntax (i.e. readability), naturalness for the application, ease of module creation and use [9].

3.1 CASL as a Process Control Language

CASL seems to be a good batch control language. The abstraction of concurrency concepts namely a heartbeat and phase, allows the CASL programmer to concentrate on the content of a module rather than on concurrency. Re-entrancy of sequence code and the ability to pass PLISTS as parameters to a sequence maps well onto the batch processing environment. A particular sequence (recipe) may be defined and by simply passing a different PLIST (set of ingredients), a different batch of a products may be produced. This together with the instantiation of multiple PLIST definitions, the interface with an Operator Communication Package, and the standard interface to the plant database, all contribute to the success of CASL as a process control language — evidenced by the more than 60 installations of CASL in South Africa.

3.2 CASL as a High Level Language

When judged as a high level language, CASL has a number of deficiencies. In the following sections we examine these short-comings and propose solutions.

Some of the unfortunate features of CASL cannot be addressed without making the new language incompatible with the old. We first discuss those issues that can be addressed while maintaining upward compatibility and then point out those to which the solutions, we believe, have more serious implications.

Local Variables

CASL allows only four local (fixed-named, and typeless) variables per job. The main reason for this restriction was the severe memory restrictions the original system had to cater for. This is no longer the case and CASL can be enhanced with proper local variables.

Proper local variables can be introduced through a special kind of PLIST, namely a LOCAL PLIST. LOCALs are declared at the start of a sequence, before the executable statements. The same format is followed as that of a PLIST declaration in the PLISTDEF section of the module. For example,

```
SEQUENCE MAINSEQ
LOCAL PLAD(16) locplad,
    INT locint1, locint2, locint3,
    MESS messloc
    :
    x1 = locint1 + locint2 * locint3
    :
```
The same usage and naming convention applies as for normal PLISTS, except for the scope of the variables. While the names of local variables must be unique, their scope are limited to only the sequence in which they are defined. Therefore, a local variable in sequence A is not visible to sequence B in the same job, and neither is a local variable in sequence A of job J visible to sequence A in job K. This is achieved by ensuring that every time a sequence is invoked, an instantiation of the local PLIST is created. This is not unlike invocation records and stack frames in conventional programming languages.

**Interface variables**

CASL allows only PLISTS to be passed as parameters to sequences. It is not difficult to extend this to allow any variable to be passed as a parameter. We call these additional parameters *interface variables*. They may be used in combination with PLISTS. For example,

```
PLISTDEF PLIST SOMEPLIST = ...  
PLISTNUM 1: SOMEPLIST(1)  
SEQNUM  
10: MAINSEQ  
20: SUBSEQ SOMEPLIST, (INT, PLAD)  
SEQUENCE MAINSEQ  
LOCAL INT COUNT  
  ;  
  CALL SUBSEQ SOMEPLIST(1), (10, COUNT, 'HV-100)  
  ;  
ENDSEQ  
SEQUENCE SUBSEQ APLIST, (INT ivint1, count, PLAD valve)  
  SET valve.MVE = 100  
  APLIST.SOMEFIELD = ...  
  ;  
ENDSEQ
```

Interface variables can be implemented as yet another type of PLIST — at each CALL or START an instantiation is made of an interface PLIST containing the values of the actual parameters. This instantiation of the interface plist is destroyed once the called sequence terminates. Interface variables are therefore passed by value.

**Looping Constructs**

CASL does not have any flexible looping construct apart from the GOTO statement and the REPEAT ... FINISH loop, which is restricted to a maximum of 255 iterations. One of the reasons for this is the fear that a loop can take long to complete (e.g. because of a large repeat count) and if it does not contain an end of phase statement, the job may overrun its time slice in the heartbeat. (A backwards GOTO forces an end of phase, therefore loops constructed from GOTOs do not have this problem.) In spite of this, it is our opinion that
more powerful looping constructs should be provided. Suitable programming practices such as putting conditional phase terminators inside loops can prevent overruns. As a compromise between simplicity and versatility, we argue for a FOR loop and a WHILE loop, namely

\[
\text{FOR } \langle \text{counter} \rangle = \langle \text{initial}_{\text{value}} \rangle \ \text{TO} \ \langle \text{final}_{\text{value}} \rangle \ \text{STEP} \ \langle \text{step}_{\text{value}} \rangle \ \text{DO} \\
\hspace{1cm} \text{NEXT} \ \langle \text{counter} \rangle
\]

and

\[
\text{WHILE } \langle \text{condition} \rangle \ \text{DO} \\
\hspace{1cm} \text{ENDWHILE}
\]

The semantics are similar to those in Pascal or Modula-2. The loop counter may be a PLIST item or a local variable. The STEP value is optional, with a default value of one. Loops may be nested.

**Real Numbers**

CASL supports a 16-bit NUM which stores a floating point number in a (non-standard) internal format. Values can range from 0.00003 to 65504.0 with an accuracy of one part in 2048. In order to obtain better accuracy and to make more efficient use of floating point co-processors where available, 32-bit IEEE format floating point numbers should be supported. Due to the current design of the intermediate language, this has the implication that all other basic data types must be expanded to 32-bit values.

**Foreign Language Procedure Calls**

CASL allows RTL/2 procedures to be called from within a sequence with the statement

\[
\text{RTL n}
\]

where \( n \) is an integer identifying the procedure to call. Parameters and results can only be passed through the 4 local variables. There is no reason why these procedures cannot be identified symbolically. Support should also be provided for other foreign languages such as C. Foreign language procedure call statements could therefore be defined, e.g.

\[
\text{XCALL C procedure}_{\text{name}} \\
\text{XCALL RTL procedure}_{\text{name}}
\]

Proper parameter passing as well as external function calls should be allowed as in

\[
\text{AREACTOR.AREA(1) = CALC.AREA(R.LENGTH, R.BREADTH)}
\]

**Symbolic Labels**

CASL allows only numeric labels in the range 0-255 since the original implementation required any label to be identifiable by a single byte. Since memory restrictions are no longer a big issue, the new language should allow the use of symbolic (alpha-numeric) names for labels.
Macros and File Inclusions

The LET text replacement facility allows for the definition of parameterless macros. Nested macros are not supported. Similarly files may be included with a file inclusion command, but only one level of inclusion is allowed. It is a relatively simple matter to expand LET definitions to allow for parameters and nested replacements and to allow nested file inclusions.

Boolean expressions

CASL does not support parenthesis to force precedence of operators in boolean expressions. This is unnecessary restrictive and simply reduces the readability of CASL programs.

All the issues discussed so far can be (and have been) addressed by implementing the proposed changes while still maintaining upward compatibility with the old language. Solutions to the issues in the following sections, however, will have more serious implications.

Layout and readability

CASL does not use an explicit statement terminator or separator like the semicolon in C and Pascal respectively. While CASL statements are always terminated by a newline character, not all newline characters terminate statements. One of the reasons given for not using explicit statement terminators [5] is that most users of CASL don't know programming languages and therefore statement terminators other than a newline would be "unnatural" for them and an unnecessary complication. We argue, however, that with the CASL syntax the way it is, the use of newline characters actually complicates matters: a set of rules is required specifying where newlines must, may, and may not be used. For example, newlines are allowed after any comma and before (but not after) ANDs and ORs in conditional expressions. It would have been much simpler to state newlines may be used anywhere where spaces can be used. It is somewhat embarrassing to have to explain to someone that she may write

\[
\text{IF } A \cdot \text{SIZE} \geq B \cdot \text{SIZE} \quad \text{AND} \quad A \cdot \text{SIZE} \geq 0 \quad \text{THEN} \quad \text{but not} \quad \text{IF } A \cdot \text{SIZE} \geq B \cdot \text{SIZE} \quad \text{AND} \quad A \cdot \text{SIZE} \geq 0 \quad \text{THEN}
\]

Had it not been for the syntax of two statements, namely the CALL and START statements, it might have been possible to allow free-format CASL programs without explicit statement terminators (while still using an LALR(1) parser). The problem lies in the fact that the two fragments

\[
\text{CALL CLEAN\_UP TANK(1) = 0}
\]

and

\[
\text{CALL CLEAN\_UP}
\]

\[
\text{TANK(1) = 0}
\]

are both legal (non-free-format) CASL fragments, but mean completely different things. In the former TANK is a PLIST with a single field that is initialised to 0 in the CALL. In the latter, TANK is an array variable (a member of a PLIST — if there is only one instantiation of a PLIST, the field name is enough to identify the variable) that is assigned the value 0.
Even without free-format the (poor) syntax of CALL and START statements give problems to an LALR(1) parser. As indicated before, PLISTS may be passed as parameters with these statements, and furthermore, they may be (optionally) initialised at that time. One could therefore have the following

```
PLIST REACTOR = PLAD OUTLET, INLET, NUM(2) REQVOL, AREA, INT ERRFLAG
PLIST BATCHREC = INT BATCHNUM, SHIFTNUM, MESS SEQTITLE, SEQID
:
CALL SUBSEQ REACTOR(1) = '2X, 'V3, 5.0, 6.0, 0.0(2), 22, BATCHREC(1)
```

which initialises the fields of REACTOR(1) to the same values of REACTOR(2) in the sample module of Figure 2. To further complicate matters, the initial values may be references to other PLIST fields, e.g.

```
PLIST BOB = INT(20) A
:
START X1 BOBSEQ BOB(1),BOB(2)=1,,3(5),BOB(4).A(2),(7), 10(5), BOB(4)
BOB(3).A = 10 % a statement following the call
```

The work of not only the parser, but also the human reader would be much easier if the initialisation of a PLIST is enclosed in braces ({} and repetition factors are enclosed in square brackets ([]). The above example could then be written as follows:

```
START X1 BOBSEQ BOB(1),
   BOB(2) = {1, , 3[5], BOB(4).A(2), [7], 10[5] } ,
   BOB(4)
BOB(3).A = 10 % a statement following the call
```

IF Statement

The IF statement has the structure

```
IF <condition> <optional_then> <statements> END
```

This allows for the following potentially confusing code

```
IF X1 = 1 % if X1 is equal to 1
  X4 = X1 % THEN assign X1 to X4
  PRINT X1 % etc.
:
END
```

The THEN should be mandatory.

LET Definition

Unlike other CASL statements, LET statements are terminated with an explicit terminator - a semi-colon. While this allows for definitions such as
LET CLEAR_FLAGS =
    X4 = 0
REPEAT 70
    X4 = X4 + 1
    FLAG(X4) = 0
FINISH;

readability can be improved if an explicit continuation character is used and no termination
character (to fit in with the rest of the language), e.g.

LET CLEAR_FLAGS = \          \          \          \          
    X4 = 0            \          \          \          
REPEAT 70          \          \          \          
    X4 = X4 + 1       \          \          \          
    FLAG(X4) = 0      \          \          \          
FINISH

String Manipulation

CASL does not support string assignments directly. In order to assign a string to a
(message) variable, one has to use a special version of the WRITE statement, e.g.

WRITE 255 MESSVAR, "New text string"

(Every WRITE statement has a priority which determines the urgency of the WRITE state-
ment. The priority range is from 0-9. The WRITE statement used with the special priority
of 255 indicates a string assignment.) It would be much simpler if one could write

MESSVAR = "New text string"

PLIST initialisation

PLIST instantiation and PLIST initialisation is performed by two separate statements
(PLISTNUM and DATA). This is an unnecessary fragmentation of information and should
rather be combined, for example

PLISTNUM
2 :  REACTOR(4) = '212XV326, '212XV315, 50.0, 60.0, 0.0(2), 22
10:  BATCHREC(1) = 0(2),
     "Transfer of raw materials for process 10",
     "X21208"

PLIST and SEQUENCE numbers

As mentioned already, each instantiation of a PLIST must be assigned a unique integer
identifier which is used at run-time to reference the PLIST items. Similarly a SEQUENCE
must be assigned a unique integer identifier by the programmer in order to reference the
sequence at run-time. We propose the introduction of a project file that controls the
execution of the project. This file would free the programmer from the need to know and
define SEQUENCE and PLIST numbers and serve to centralise the definitions and declaration
<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution implemented</th>
</tr>
</thead>
<tbody>
<tr>
<td>Four fixed-named local variables</td>
<td>General local variables via LOCAL PLIST.</td>
</tr>
<tr>
<td>Only PLISTS as parameters</td>
<td>Interface variables</td>
</tr>
<tr>
<td>Limited looping construct</td>
<td>FOR and WHILE statements</td>
</tr>
<tr>
<td>Accuracy of real numbers</td>
<td>32-bit IEEE format floating point numbers</td>
</tr>
<tr>
<td>Only numbered RTL procedure calls</td>
<td>Support for symbolic calls to C procedures</td>
</tr>
<tr>
<td>Only 256 numeric labels</td>
<td></td>
</tr>
<tr>
<td>Limited LET definitions</td>
<td>2^32 symbolic labels</td>
</tr>
<tr>
<td>Boolean operator precedence</td>
<td>Parameters; nested invocation; nested file</td>
</tr>
<tr>
<td>Lack of free format</td>
<td>includes</td>
</tr>
<tr>
<td>START and CALL statement syntax</td>
<td>new syntax</td>
</tr>
<tr>
<td>Optional THEN in IF statement</td>
<td></td>
</tr>
<tr>
<td>Multiline LET with terminator</td>
<td></td>
</tr>
<tr>
<td>No string assignment</td>
<td></td>
</tr>
<tr>
<td>PLIST instantiation and initialisation</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Summary of Improvements

of data. Currently every module is required to include a PLISTDEF and PLISTNUM section to define and declare those PLISTS that are used in the sequences contained in the module. The SEQNUM section is required to associate the sequence with a number and provide a prototype against which to check usage of that sequence.

The project file would contain all PLIST definitions and declarations for the entire project as well as all the SEQUENCE prototype definitions. It would then be the compiler's responsibility to assign appropriate numbers. Individual modules would be compiled against the project file and the system would ensure that dependent modules are recompiled as appropriate.

4 Implementation Issues

A new CASL compiler has been developed that implements the improvements described before that could be made while maintaining upward compatibility. Figure 3 summarizes the improvements identified and implemented. Those that were not implemented were either because of the compatibility issue, or because the initiators of the project saw no immediate need for it.

In the rest of this section we describe some of the problems that had to be solved when implementing the compiler using PCLEX and PCYACC [1].

4.1 Language Specification

Initially an attempt was made to make the language free-format without explicit statement terminators. However, as explained in a previous section, this was not possible. Newlines were therefore encoded as statement terminators, with the same restrictions and exceptions as were present in the original CASL.
As has been indicated before, the complete START and CALL statements cannot be implemented using an LALR(1) parser (which is what PCYACC produces). In particular, the problem is when PLISTs are initialised in these statements. Fortunately it turned out that in practice, START and CALL statements are not often used with PLIST initialisation. This, coupled with the desirability of a grammar-based parser was judged to be a good enough reason to change the syntax of these initialisations to the format described earlier. This is the only case in which the new language is not upwardly compatible with the old.

4.2 Preprocessor and Lexical Scanner

PCLEX was used to generate a lexical scanner from regular expressions defining the CASL tokens. An attempt was made to incorporate file inclusions and macro definitions and calls directly into the scanner and parser. This could be achieved by changing the source of the scanner input at appropriate times. However, because PCLEX generates a scanner that uses the more efficient, so-called flex algorithm, this was not possible. Instead of reading input a character at a time, a flex scanner reads input a line at a time and then examines the characters in the internal buffer. Redirecting input source under these circumstances became too complicated and cumbersome, and a separate, hand-coded preprocessor was developed using a public domain C preprocessor as a base.

A minor problem arose during the implementation of the lexical scanner. In CASL two or more strings separated only by a newline are considered as one string. A string token was therefore defined accordingly. However, this caused the scanner's internal buffer to overflow on long strings. The buffer size of the scanner could have been extended, however, one would not be able to guarantee that it is big enough since there is no limit on the lengths of strings (in some applications strings of more than 1K bytes are defined). The problem was solved by moving the recognition of long strings to the parser. Rules were defined to recognise lists of strings only separated by newline characters and to concatenate these strings and store them as a single string.

Tag name to Plant Address conversion

The convention for referencing plant addresses in CASL is by tag name, which is usually application dependent. The standard CYGNUS syntax requires that a tag name begin with a prime (') followed by up to nine significant characters, drawn from the letters A–Z, digits 0–9 and @. The layout characters - and / may be used freely, e.g. 'FIC-410/A, 'QMASSFLOW, and '77-HV-101/A. A tag name that is used in a CASL module must be present in the CYGNUS database in order to be valid. Since each PLIST item is represented internally by a 32 bit value (16 bits in the original CASL), tag names, which are the "values" of PLAD variables, need to be converted to 32 bit values. In other words, a PLAD is a 32-bit representation of a tag name.

In the old version of the CYGNUS system the conversion from tag name to PLAD, was performed by the translator. This was achieved by simply passing the tag name to a special dictionary task, which looked up the name in a dictionary and returned an associated 16 bit number that served as a kind of index in the database. This number was then stored in the PLAD item and used for subsequent database access. The disadvantage of this approach is that if the database is changed, all modules referencing the database have to be re-translated.

In the new compiler, the tag name to PLAD conversion is left to the loader, to be performed at load time. The compiler records every occurrence of a tag name and passes
this information to the loader. Multiple occurrences of the same tag name will result in a chain of pointers being formed from the last occurrence to the first. The loader converts the tag name and inserts the packed value into the intermediate code. The loader thus becomes responsible for the validation of tag names.

Run time conversion and validation is not required since there is a restriction that prevents the database from being updated while a sequence that uses it is running.

5 Conclusion

We have examined CASL, a specialised high level language for batch control, and concluded that, while it serves well as a batch control language, it could do with some improvements as a high level language. We have identified specific areas for improvements to functionality, clarity and readability and have proposed specific improvements. The majority of these improvements can be implemented while keeping the new language upwardly compatible with the old. Finally, we have implemented an LALR(1) compiler that incorporates all the upwardly compatible improvements. Except for one infrequently used construct, the new compiler will compile all existing CASL code in order to protect the large investments made in CASL.

References


[2] AECI Process Computing, P.O. Box 1823 Halfway House South Africa. CYGNUS PROCESS AND MANAGEMENT SOFTWARE.


Design of an Object-Oriented Framework for Optimistic Parallel Simulation on Shared-Memory Computers

Philip Machanick
Department of Computer Science, University of the Witwatersrand, 2050 Wits

Abstract

Parallel simulation, if it is to become a mainstream technology, must become reasonably accessible to programmers without unusual skills. Since low-cost shared memory machines are becoming an increasing possibility, a simulation package designed to perform efficiently on such machines is desirable. This paper presents previous results indicating some performance issues to be taken into account in such a simulator, and presents a design for an object-oriented package, based on features of C++, for a simulator using the optimistic model. The major finding is that C++ is suitable for such a simulation package, although there are a few difficulties are identified.

keywords: Simulation, Parallel Processing, Object-Oriented Programming
Computing Review Categories: I.6.1, C.1.2, D.2.2

1. Introduction

Parallel simulation is becoming an increasingly important area, as demands for simulation increase and parallel machines become more affordable. At the same time, if it is to be useful in organizations with a relatively low budget, it should not be significantly more difficult to program than sequential simulation. The starting point for the work reported on here is an investigation of techniques for implementing software efficiently on shared-memory machines, which rely on good caching behaviour to achieve scalability. Such machines are likely to become increasingly common—and affordable—in the future, as the speed gap between affordable high-performance processors (typically RISC) and affordable memory components increases [5]. The Silicon Graphics Iris series is an existing range of machines approximating to the predicted features. There are also strong indications that Sun is to launch a machine in this class soon.

The approach taken here is to design a framework for a specific model of parallel simulation, using object-oriented techniques to present a high-level interface to the simulation programmer.

This paper builds on experience of implementation of a specific parallel simulation in the object-oriented language C++, and presents a design for a general object-oriented library for such simulations. In the previous work [4], the Argonne National Laboratories (ANL) macros [17] were translated to C++ (making minimal changes). This approach was not entirely successful in that the macros are language extensions outside of the compiler, and debugging could be difficult (for example, it was generally necessary to read the expanded macros to interpret error messages). In addition, the macros are general parallel constructs, rather than simulation constructs, and in this sense present a low-level interface to the programmer.

One of the purposes of this paper is to demonstrate that C++ is a sufficiently powerful language to achieve the required functionality without extensions (either via macros or compiler changes). Another purpose is to present a design of a general package for parallel simulation primitives in an object-oriented language, based on performance-oriented research results. Much previous work on parallel simulation has been on unconventional architectures, such as vector processors [18], the Connection Machine [6; 3], Hypercubes [14] and the BBN Butterfly [16; 22].

The rest of this paper presents some background, followed by the design. The next section outlines general development of parallel simulation, with special emphasis on the optimistic model. The following section reports previous work in implementing a parallel simulation in C++. The design follows in the next section. This is followed by a summary of related work. In conclusion, some findings are presented, along with plans for future work.
2. Parallel Simulation

In this paper, 3 models of parallel simulation are considered: conservative, optimistic and time-stepped. Of these, the time-stepped simulations are the least interesting from the point of view of research, because they are relatively easy to implement. The conservative and optimistic methods are both event-driven, and are essentially in competition, while the time-stepped method is usually applicable to different problems. However, some insights derived from research into a time-stepped simulation have proved useful as a basis for this work.

Time-stepped simulations generally proceed by processing a large amount of data at specific discrete time intervals. They may be implemented on a range of architectures reasonably simply. On vector machines, they are implemented by scheduling all the computations of a given type so they can be done simultaneously to maximize vectorization [18]. On a SIMD machine such as the Connection Machine, the work can be distributed reasonably easily over the whole machine [6]. On a shared-memory or distributed memory machine, they can be implemented using barriers to synchronize time-steps.

The conservative method is a distributed implementation of a conventional event-list discrete-event simulation [19]. In the conservative model, each logical process (LP) in the system being simulated is mapped onto a process in the program, and the event list is distributed between LPs. A communication network between the processes supplies synchronization and data transfer. Each LP lp has a clock value \( T_l \) associated with it, which is the minimum timestamp of the most recent event it has received from each of its neighbours. A process blocks if none of the events in its local event list has a timestamp earlier than \( T_l \).

A big problem with the basic conservative method is that it can deadlock. There are various techniques for deadlock detection, such as sending null messages which have no semantic value, but serve to advance the clock. A conservative simulation can be flooded with null messages. Some techniques exist to reduce the number of null messages, such as cancelling them when another message with a higher timestamp (from the same LP) is encountered [20].

Another problem with the conservative method is that there may be considerable time lost as a process blocks waiting for its neighbours. One approach to reducing this problem is the use of lookahead, in which a given LP knows the latencies, service times etc. of its neighbours and can predict how far into the future they will send a message if they have not as yet processed one. Lookahead can give good results [25], but has the drawback that it generally requires application-specific knowledge [11].

The optimistic method [13] avoids deadlocks and unnecessary delays by allowing an LP to continue processing even if it is ahead of its neighbours in simulation time. Each LP operates in its own virtual time, and the minimum virtual time over all the LPs is called global virtual time (GVT). If a message called a straggler is received which has an earlier timestamp than one already processed, there is a causality error, and any invalid work has to be undone. This is done by sending out antimeessages, cancelling the effect of erroneous messages. This cancellation is effected by rolling state back to the time prior to the cancelled message. Old state has to be maintained to make rolling back possible. Saved state which is older than GVT is no longer needed, and is referred to as a fossil. Fossils must eventually be reclaimed, when memory becomes scarce; this is known as fossil collection. One proposed relatively low-cost approach to fossil collection is to clear the dirty bit in the memory manager's page table when a whole page is known to contain fossils. Such a scheme is being investigated in new work on at least one experimental operating system (though from a different starting point): the V operating system [2].

The mechanism used to realize virtual time using antimeessages and rollbacks is called time warp.

The intention is that time warp will use up what would otherwise be idle time, or deadlock detection or avoidance, in a conservative simulation—but it is possible that the overall cost will be higher than the saving. Analytical results show that the transition between cases where time warp is a net win and examples where it is not (when rollback costs more than is saved) is smooth [9]. The implication is that techniques to reduce the amount of rollback are likely to to push the technique close to the performance of conservative methods, even in cases more favourable for conservative simulations. Some work has already been done to limit rollback, by placing a limit on how far any LP can go ahead of GVT [23]. There is however some doubt about the value of this approach, called moving time window (MTW), because it is non-discriminating in holding up LPs, regardless of whether the work they are doing "ahead" of GVT will turn out to be useful or not [11].

Another variation is the replacement of the aggressive cancellation strategy by lazy cancellation, in which antimeessages are only sent

---

out when it is determined that previous computations did in fact produce erroneous results. This approach results in a performance improvement in most cases, despite the extra overhead of having to check if the state has changed before deciding to cancel [21].

A recent proposal, \textit{temporal decomposition}, is to allow an LP to be scheduled as a different process after a specific point in virtual time. The break between two versions of the LP is handled conservatively, in the sense that the later version is only allowed to start after there is no chance of rolling back to the previous version. This strategy allows for dynamic adjustments to load balance, even in a statically load-balanced system, by allowing the LP to be assigned to different processors before and after the split [22].

One drawback of optimistic methods is the difficulty in keeping track of state which may potentially need to be rolled back, in a general programming model with arbitrary side-effects [11]. There is a case for a higher level model to support state manipulation, to ensure that state is not changed without the knowledge of the rollback mechanism. Another problem is some categories of events cannot be rolled back (for example, if they result in a state change external to the simulation, such as output to the screen). The original time warp mechanism special-cases such events [13]. The \textit{nonretrac{}table event} (NRE) is a useful addition to the model. Such events cannot be rolled back, and so are only scheduled when GVT has caught up with them. Aside from providing a model for events which cause external state changes, they can also be used to implement flow control, including controlling the degree of optimism [1]. In the limit, if all events are nonretractable, the simulation becomes sequential.

There is considerable scope for optimizing both conservative and optimistic methods, so that both come closer to solving problems best suited to the other (e.g., Wagner's paper [25] was a rebuttal of a claim that a specific problem was only suited to the optimistic method). However, the argument that the optimistic method is less dependent on application-specific optimizations [11] is attractive.

For this reason, the new work introduced in this paper is oriented towards the optimistic method. Nonetheless, the experience being drawn on is from work with a time-stepped method; there is sufficient overlap of issues to make this valid.

3 This is worse than conservative, since a conservative simulator uses neighbours to determine the local clock, whereas an NRE is held up until the \textit{global} minimum clock has caught up with it. A weaker condition on when the NRE may be scheduled is possible, but is likely to lead to a requirement for the kind of mechanism found in conservative simulators to prevent deadlock.

3. Experience with a Particle-Based Simulator

This section introduces aspects of previous experience with parallel simulation, focusing on aspects relevant to the design presented in this paper. In particular, the overall program structuring techniques which are designed to achieve good caching behaviour are of relevance.

The application is MP3D, a particle-based simulation of a wind tunnel. The simulation was originally implemented on a Cray 2 [18], and the version initially used had been ported to an Encore Multimax with relatively few changes from the vector version. This Encore version has been used as a benchmark example of a program with poor caching characteristics on shared-memory machines [12]. This initial version is called MP3D-0 (actually with minimal changes from the Encore version to convert it to C++).

Each timestep, particles are moved (depending on their velocity components), and paired for possible collision. A probabilistic selection function is used to determine whether the collision actually takes place. The $n^2$ problem of finding collision partners is reduced to $\Theta(n)$. The strategy is to divide space into unit-cube cells. After particles are moved, they are randomly paired within a cell, rather than attempting to find nearest neighbours [18]. In MP3D-0, randomization is achieved by having processors contend for particles to process, both in the move and collide phases, and by randomly ordering particles within the array in which they are stored.

There are three major weaknesses in the organization of MP3D-0. The first is the fine-grained approach, which makes poor use of caches, and the second is its data structures (following the organization originally used for the vector implementation) are in the form of several arrays, each containing a single attribute of a particle, for example, its $x$ spatial co-ordinate. The final problem is the random positioning of particles in arrays, which makes for poor locality.

The first improvement made to MP3D was to group related data together into objects, which are padded and aligned to fit cache blocks. This modification has the effect of avoiding \textit{false sharing}, in which the same cache block contains unrelated data items, while increasing the \textit{prefetch} effect of large cache blocks (all the data relating to one particle may be fetched after a single cache miss). The resulting program, MP3D-1, has improved caching behaviour, but is still poor.

4 These changes have no measurable impact on performance, though the code size is increased, owing to the larger C++ libraries.
The second improvement was to reorganize the program around the spatial locality inherent in the model. The resulting revision, MP3D-2, is a major restructuring of the program. Each processor is given a fixed area of space, containing a collection of cells called a *precinct*. Each timestep, a processor moves all its particles, interchanges those which are no longer in the same precinct with its neighbours\(^5\) and then does collisions. Because the randomness of association between particles and processors can no longer be relied upon to ensure collision partners are randomly paired, randomization is done explicitly. Each particle contains the necessary information to find the cell it belongs to (needed once it has been moved), and each cell contains the identity of the precinct that owns it. The key aspect of this implementation is that data stays with the same processor as long as possible, thereby avoiding the high number of cache invalidations of the previous implementations.

MP3D-2, while a specific restructuring exercise, is a basis for evaluating some general techniques, and the suitability of C++ for implementing them. Some of these strategies are:

- **Increasing processor affinity**—ensuring objects and groups of related objects are processed by the same processor as far as possible, rather than reassigned in a very fine-grained way.
- **Separation**—the forcing of unrelated data into separate cache blocks, thereby minimizing false sharing.
- **Co-location**—placing data which will be accessed at approximately the same time and with similar read-write characteristics (e.g., accessed by one processor) in the same cache block, thereby enhancing the prefetch effect.

These strategies have been realized in MP3D-2 by two major techniques: the use of a *space directory* to support the relating of objects to the precincts (which correspond to a unit of space allocated to a processor), and the use of overloaded memory allocators, which replace the standard C++ allocators.

Simulations show that MP3D-1 has a miss ratio about 5 times lower than MP3D-0, while MP3D-2 improves over MP3D-0 by at least an order of magnitude. The significance of these changes depends on a number of factors, such as the fraction of references which are to shared memory. However, with a tendency is towards greater penalties for cache misses (up to hundreds of processor cycles), and large cache blocks [15], such changes in caching behaviour are likely to become increasingly significant. Some designs of fast uniprocessors already have cache block of 128 bytes [2], and this is a trend which is expected to continue [15].

It is possible to measure the approximate effects of the differences between the programs by measurements on machines currently available, even if they do not exactly match these characteristics.

Machines on which measurements have been made are a DEC Firefly with MicroVAX CPUs, a Firefly with essentially the same memory system but faster CVAX processors, a 4-processor Silicon Graphics SGI 4D/240S and a faster 4D/380S with 8 processors. The improved program structuring shows up in improved speedup, and more effective use of faster processors. Figures [4] supporting these claims appear in Table 1. Speedup is given relative to the uniprocessor version of the same program, as we are concerned with scalability rather than with algorithm analysis. Speedup for faster processors is given relative to the equivalent run on a slower version of the same machine.

The Silicon Graphics is a good example of the modern trend in shared-memory machines. It has fast MIPS CPUs, 16-byte cache blocks\(^6\) and a relatively low memory bus bandwidth in relation to the speed of the processors. The Firefly is an older design. Its cache blocks are only 4 bytes (i.e., too small for false sharing to be an issue), and the purpose of the caches is more to reduce bus traffic than to reduce memory latency [24]. Despite these differences between these machines and from expected future architectures, the restructuring of MP3D produces good results on both types of machine, suggesting that the proposed techniques are reasonably general.

### 4. A Design

The strategy is to start with requirements for optimistic simulators, and to work towards the insights derived from the MP3D exercise (the low-level of the design).

The general goals for the design are transparency, efficiency and extensibility. The intention is to provide a general framework for optimistic simulations, incorporating the most common features, while making allowance for the addition of new features. Efficiency is to be achieved by low-level optimizations, which are hidden from the programmer, using the information-hiding capabilities of an object-oriented language.

---

\(^5\) The simulation is constructed so that particles will not move more than one cell, so it is possible to determine in advance which precincts are "neighbours".

\(^6\) Not as large as some more recent designs, but large enough for false sharing to be an issue.
<table>
<thead>
<tr>
<th>machine and number of processors</th>
<th>MP3D-0 s sp.r sp.f</th>
<th>MP3D-1 s sp.r sp.f</th>
<th>MP3D-2 s sp.r sp.f</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGI 4D/240S</td>
<td>1 677</td>
<td>2 503 1.3</td>
<td>4 407 1.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>597</td>
<td>410 1.5</td>
<td>283 2.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SGI 4D/380S</td>
<td>1 544 1.2</td>
<td>2 455 1.2 1.1</td>
<td>4 409 1.3 1.0</td>
</tr>
<tr>
<td>(faster CPUs)</td>
<td>437 1.4</td>
<td>315 1.4 1.3</td>
<td>268 1.6 1.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>569 1.2</td>
<td>312 1.8 1.2</td>
<td>179 3.2 1.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MicroVAX Firefly</td>
<td>1 12839 1.3</td>
<td>2 8096 1.6</td>
<td>4 5818 2.2</td>
</tr>
<tr>
<td></td>
<td>12700</td>
<td>8389 1.5</td>
<td>5810 2.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>9567</td>
<td>5116 1.9</td>
<td>2732 3.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVAX Firefly</td>
<td>1 4854 3.9</td>
<td>2 4478 1.1 1.8</td>
<td>4 4002 1.2 1.5</td>
</tr>
<tr>
<td>(faster CPUs)</td>
<td>4161 3.1</td>
<td>3964 1.0 2.1</td>
<td>3632 1.1 1.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3303 2.9</td>
<td>1790 1.8 2.9</td>
<td>1014 3.3 2.7</td>
</tr>
</tbody>
</table>

Table 1. Run time and speedup: 3 versions of MP3D on 4 machines
s is time in seconds, sp.r is speedup with relative to the uniprocessor case and sp.f is speedup of an equivalent machine with faster processors.

The features in the initial implementation are chosen from those which have been generally accepted, with allowance for addition of some of the less tested innovations.

State changes should generally be made through state manipulation primitives, to ensure that the rollback mechanism is reasonably simple. Nonretractable events are supplied to allow a back door for more general state changes. Although they are a relatively untied concept, a similar mechanism would in any case have to be supplied to handle input and output. In the initial model, aggressive cancellation is used, and lazy cancellation is left as an extension. Moving time window is not implemented in the initial model, because there is uncertainty as to its usefulness. There is in any case potential for investigating the claims for nonretractable events as a throttling mechanism for excessive optimism (since NREs are needed anyway).

The remainder of this section presents a high-level view of the design, the approach to implementation in C++, some C++ techniques and an outline of the resulting classes.

4.1 high-level design

The structure of the simulator is broken down into LPs. Each LP contains an input queue, containing events not yet processed, stored in timestamp order. State associated with the LP is locally stored. An event dispatcher schedules and executes events in timestamp order, and puts events to be passed to neighbours in an output queue, and makes copies of the events in the antimessages queue. When a straggler arrives, the dispatcher passes control to the rollback mechanism, which decides which antimessages should be scheduled for transmission and local execution. Those for transmission are sent to the neighbours, and those for local execution are used to determine how to roll the local state back.

The relationships between the components of an LP are illustrated in Figure 1.

LPs are grouped together, to maximize spatial and temporal locality. Each such group is assigned to a single processor, in the style of the precincts used in MP3D. Load balance is more complicated to determine in an optimistic simulator than a pessimistic or timstepped simulator. As long as an LP does not run out of internally generated events, or stick on an NRE, it will not be idle. However, it is undesirable that a single LP should get arbitrarily far ahead, and flood the network with anti-messages. A definition of load balance in terms of the distance an LP is ahead of GVT is one possibility. This is not ideal, for the same reason that moving time windows are potentially a problem: the fact that an LP is far ahead is not necessarily a problem, if all the work it is doing is valid. However, changing the load balance will not result in idling a processor unnecessarily, as happens in MTW.

One possibility for dynamic load balance is a temporal decomposition scheme. However, this is overkill on shared-memory machines, as transferring state to another processor is accomplished relatively cheaply by passing a pointer, and allowing the caching mechanism to transfer data on demand (saved state for rollbacks in particular will not be transferred if it's not needed). Something very similar to temporal decomposition can in any case be achieved by scheduling an NRE as part of the protocol for moving an LP to a new precinct.
This NRE would in effect force the LP to wait until GVT had caught up with it before it was transferred, ensuring that no checkpointed state would need to be copied.

The details of the load balance process will be empirically determined, based on performance measurements. Figure 2 illustrates grouping of LPs in precincts, and how this may change dynamically. If a precinct is “too conservative”, it will cause a lot of rollback in its neighbours (which may go too far ahead of GVT). Achieving a good load balance has to take this into account as well as simply ensuring the processors are equally loaded.

There is a general-purpose utility process, which does work such as checking load balance and periodically recomputing GVT. This process is executed by any processor which is idle, or after a new value of GVT is required (for example, to do fossil collection). This process is available to each precinct, and can be run by any of them.

4.2 some detail: the C++ approach

To implement these ideas in C++ (taking into account the findings of the MP3D research) it is useful to divide classes into several layers, representing the hierarchy of abstraction, from the simulation programmer’s view down to machine-specific details. There is a hierarchy of classes, with inheritance links through each of these layers. An overview of the class hierarchy and its relationship to the layers is shown in Figure 3.

The layers (with examples of the classes defined at each layer) in the design are:

- **kernel**—these are the machine-specific details, at the base of the underlying threads package, such as locks, memory allocators and process creation and termination. Classes at this level include *Object*, which encapsulates low-level memory allocation and deallocation, *Lock*, which locks a specific named lock object until the end of the scope in which the Lock appears and *Process*, which launches a new process.
- **low_level_simulator**—low-level implementation details of the simulation mechanism, including time warp, message-passing between LPs and generic versions of messages. Classes at this level include *Precinct, LP, Generic_queue, Saved_state*, and *Antimessage*.
4.3 some C++ techniques

Inheritance plays an important role in the design. One example is the implementation of antimessages. An antimessage is a part of the low_level_simulator layer, and therefore cannot be redefined by the programmer. However, class Antimessage contains a pointer to a Message object, which may be a derived class created by the programmer, rather than the original class Message. As long as the member functions (C++ terminology for methods) needed by Antimessage have not be redefined in an inconsistent way, Antimessage will function as intended.

One of the ideas used in various parts of the design is the notion of defining\footnote{In C++, as in C, a name may be declared, in the style of a forward declaration in other languages such as Pascal. A full declaration is called a definition.} an object, the sole purpose of which is to generate correct initialization and termination, using the implicitly called constructors and destructors. This technique has been used for implementing tracing [7]. The general idea is an object is defined, resulting in a call of its constructor at the place of definition. At the close of the scope where the definition occurs, the destructor is automatically called. For example, in the C++ fragment:

\begin{verbatim}
{ Simulation_environment sim;
  // code to do simulation
}
\end{verbatim}
the object sim exists between its definition and the 
the close of scope symbol, ). The purpose of this 
construct is to call the constructor for 
Simulation_environment objects, which does some 
global initialization. The destructor—automatically 
generated by the compiler—ensures termination 
code is called, which is useful to ensure that calling 
the terminating code is not forgotten by the 
programmer. A similar approach is used for 
implementing locks (though these are not expected 
to be used by simulation programmers).

Each class may contain static data members 
(class variables, in Smalltalk terminology). This is a 
useful feature for avoiding the use of global 
variables. In the design, this feature is used in two 
contexts: memory management and maintaining 
global state. The low-level memory management 
routines, defined in the kernel, have the option of 
taking a free space list as an argument. Where this 
feature is used, the free space list is stored as static 
data member in the class defining the specific 
allocators and deallocators.

Optional parameters make it possible to use the 
same low-level allocators, whether the programmer- 
supplied free space list is to be used or not.

Another useful feature is capability of 
overloading of the array index operator, [ ]. This 
makes it possible to a variable-sized 3-dimensional 
array, with same syntax as a statically declared one. 
The first two indexing operations return slices of the 
space directory, with decreasing dimensionality 
each time, and the third indexing operation returns 
the element desired. This capability would be useful in 
simulations where spatial relationships between 
LPs are a simple function of their positions in space, 
as in MP3D.

4.4 some C++ classes

This section presents an outline of the external 
interfaces of the classes.

The kernel defines the following classes:

• Object—defines low-level memory 
management, including alignment and 
padding to fit cache blocks.
• Lock_data—defines data structures to 
implement locks.
• Lock—the constructor for this class sets a 
lock (identified by an object of class 
Lock_data), and the destructor releases the 
lock at the close of the current scope. A 
typical example looks like this:

```cpp
{ Lock output(output_lock);
  // code to do some output
} // lock released here
```
• Process—its constructor launches a thread, 
calling the function passed to it as a 
parameter, and its destructor cleans up after 
process termination.
• list manipulation and other generic data 
structure primitives, such as dynamically 
resizable 3-dimensional arrays.

The low_level_simulator level defines the following 
classes:

• Precinct—a precinct contains a list of LPs, 
and a scheduler to decide which to execute 
next. It also knows which other precincts are 
its neighbours, and manages message 
exchange between itself and them.
• LP—a logical process; this relies on the 
semantics of Message and State to implement 
a specific simulation, and Precinct defines 
communication with LPs on other processors.
• Saved_state—contains list of changes to state, 
including identities of messages that caused 
the changes, stored as antimessages; the 
rollback controller is contained in this class, 
as is the fossil collector.
• Anti_message—contains a copy of the original 
message, plus its own timestamp.

Classes defined in user_level_simulator include:

• State—includes state change primitives and 
recording changes for rollback. The 
programmer will have to derive a new class 
from this one to incorporate the semantics of 
the state for the actual simulation problem.
• Simulation_environment—one object of this 
type is defined at the start of the program. As 
with Lock, its constructor and destructor 
respectively initialize the program for 
simulation, and clean up at the end.
• Message—contains a timestamp and 
identification of its originating LP. Uses 
member functions (methods in conventional 
object-oriented terminology) in State to make 
changes to the state when its own execute() 
member function is called.
• NRE_message—a nonretractable event 
message: may make arbitrary state changes, 
including doing input and output; not 
scheduled until the precinct detects GVT has 
passed its timestamp.
5. Related Work

A number of object-oriented threads packages have been implemented for shared-memory parallel machines. An example is Presto which has been implemented and evaluated on a 20-processor Sequent Symmetry [8]. Work in this category has mostly aimed at efficiency of low-level primitives, such as locks and launching threads. In this work, the emphasis is more on support for application structuring to achieve good caching behaviour.

Other work in implementation of optimistic parallel simulators on shared-memory parallel machines has not addressed the locality and caching issues raised here. Examples include BBN Butterfly implementations of optimistic simulations [11]. In terms of its programming model, the Synapse simulation toolkit has similar higher-level goals to the design presented here, except it supports the conservative model [26]. Synapse is built on Presto, and therefore does not address new implementation issues at the lower level.

In other work on structuring simulators, attention has been paid to the difficulties of efficiently vectorizing object-oriented code in C++ [10]. This is in contrast to the results on which this work is based, in which object-oriented programming is shown to be suited to cache-based architectures [4].

6. Conclusions

The design as presented here has been partially implemented (mainly the kernel level).

The goal of showing that C++ is a sufficiently powerful language for such a package has been partially met. All the required features are possible to implement, some conveniently and reasonably simply. For example, the inheritance mechanism makes it possible to hide the antimessage mechanism from the programmer, while allowing antimessages to use new classes derived from the standard Message class.

There are however some problems. The most serious is that the requirement that messages should not make state changes except through the State class, to make rolling back manageable. There is no way this rule can be enforced through the language; a preprocessor or modifications to the compiler
would appear to be the only sure ways of enforcement. Since there is a convenient mechanism for implementing general side-effects through NREs, it is possible this limitation will not cause major problems in practice.

Another deficiency is that the class hierarchy does not exactly match the logical breakdown of the problem, because some features which are logically low-level (such as antimessages) need information from higher-level constructs (messages, in this case). Although the language makes such an implementation possible, this situation makes it more difficult to describe the logical hierarchy than is desirable.

The next phase of this research is the implementation of the simulation framework described here. Experience with using it to implement simulations will be the basis for evaluating how serious these problems are.

In addition, performance measurements on both standard benchmarks and real problems will be made investigate validity of the overall program structure, especially the aspects which were successful in the MP3D exercise.

C++ has been successful as a basis for one parallel simulation exercise. The design presented here indicates that it is a promising basis for a more general package. In particular, the potential for designing a parallel simulation package that is usable without deep knowledge of the underlying technology, including parallel programming and the mechanisms of optimistic simulation, is attractive. This design is support for the view that such a framework is feasible.

7. Acknowledgements

I would like to thank David Cheriton for supporting the underlying work on which this paper is based. Henk Goosen also played a leading role in the architecture-related aspects. András Salamon made helpful comments on an earlier draft. The work on which this paper is based was supported in part by DARPA Contract N00014-88-C-0619, the Anderson-Capelli Fund and the Mellon Foundation.

References

Multiconference on Distributed Simulation, January, 55–62.


Using Statecharts to Design and Specify a Direct-Manipulation User Interface

 Lynette van Zijl and Deon Mitton
 Institute for Applied Computer Science,
 University of Stellenbosch, 7600 Stellenbosch,
 Republic of South Africa

Abstract

Statecharts were developed by Harel et al [10] to specify complex reactive systems. In this paper we report on our application of statecharts as a design and specification tool for an X-Windows based Graphical User Interface for the telephone network performance modelling tool GMA.

1 Introduction

The design and specification of a user interface is often perceived as a straightforward form layout task. However, now that graphical direct-manipulation interfaces have become the norm, the complexity of the design, specification and coding of the user interface equals that of any other system component. This complexity led to a renewed interest in the specification of user interfaces over the last five years[4]. At the Institute for Applied Computer Science (ITR), where we develop commercial performance evaluation packages, we experienced the need for a modern specification methodology for direct-manipulation graphical user interfaces.

Before we discuss specification methodologies for user interfaces, let us first recapture on the structure of a typical user interface. One of the best-known models for a user interface is the Sehein model[6]. In this model, the interface is conceived as having three components: The Presentation Component, the Dialogue Component and the Application-Interface Component (see figure 1). The Presentation Component concerns the visual aspects of the interface as presented to the user; the Dialogue Component concerns the dialogue between the user and the interface; and the Application-Interface Component concerns the connection between the application and the interface. Issues relating to the Presentation and the Application-Interface Components are less well known than the issues relating to the Dialogue Component, for which a number of specification methodologies have been proposed.

Specification methodologies can be loosely classified as being either formal or informal, and either being graphical or non-graphical. There are valid arguments for and against the choice of any of these languages; we opted for a formal graphical specification language (see section 3 for our motivations). Graphical specification languages that are based on state transition diagrams are powerful yet easy to understand and use. However, state transition diagrams suffer from two major problems when used to specify user interfaces. The first problem is the state explosion that occurs in any sizable system, and the second problem is the inability of the standard transition diagram to model concurrency. Many extensions/adaptations to state transition diagrams have been suggested in attempts to overcome these problems. One such adaptation is Harel’s statecharts (see [10]).

We applied the statechart methodology in the design and specification of a large direct-manipulation interactive user interface. In our experience we found that Harel’s statecharts overcome the problems in transition diagram specifications, in that they are suited to the specification of user interfaces based on asynchronous events (such as the industry standard X-
Windows system). Additionally, we found that statecharts can be applied to design all three of the Seeheim user interface components. As such, the statecharts provide a unified approach to interactive user interface design and specification.

The rest of this paper is organised as follows: Section 2 provides a brief introduction to the syntax and semantics of statecharts. Section 3 sketches the background to the project for which the user interface has been designed, and section 4 contains a discussion of the actual design and specification (with ample examples).

2 Statecharts Revisited

Statecharts were developed by David Harel in the mid 1980s, and are described in a number of papers[10, 12, 13]. Harel developed the statecharts as an answer to the challenge of Green[8] who claimed that there are no adequate graphical specification tools available to model interactive behaviour. Finite state machines and their transition diagrams, for example, are easy to understand, but suffer from an exponential state explosion problem.

Harel developed the generalised concept of a higraph[11] which is a diagramming object that combines the properties of hypergraphs and Venn diagrams. The higraph can therefore be used to represent a set of elements (with a special relation on them) and then to represent a collection of such sets with structural relationships among them. The interested reader can consult [11] for more information concerning higraphs; let it suffice at the moment to note that the higraph is a formally defined mathematical object, with a visual representation of its topological characteristics. The statechart is just an application of a higraph; it is essentially a higraph-based version of a finite state machine.

Since the statechart is based on a finite state machine, it is empowered with all the ergodicity of a state transition diagram. Moreover, it solves the state explosion problem found in finite state machines by having the properties of depth and orthogonality. The depth property allows for hierarchical modelling by clustering or refinement, so that atomic elements can be clustered graphically into a single composite element, or a composite element graphically refined into several smaller elements. The orthogonality property can be used to model independence and concurrency, by combining elements graphically in a representation of their Cartesian product.

Before illuminating these concepts with an example, let us first highlight the graphical symbols used to construct a statechart. Statecharts graphically represent the states and events in a system. The symbol used for a state is a rounded rectangle, with the name of that state inside the rectangle. Hierarchies of states are represented by enclosure of states inside outer states (see figure 2). Events are represented by arrows, with the arrow labelled by the name of the event and any condition which may guard that event. There are different types of arrows
available. The standard arrow is that which originates on the perimeter of one state and ends on the perimeter of another state (see Event-1 in figure 3). The default entry state is labelled by a default arrow (such as Event-0 in figure 3). The hierarchical arrow indicates that an event originates in a lower level state inside the current state. For example, in figure 3 State1 is an hierarchical state with substrates on a lower level. Here Event-2 originates from one of the states within State1 and not from State1 as such.

Now that we have the symbols for a statechart available, let us use them to model a small example. Suppose that the system to be specified is a chocolate vending machine called C (the original vending machine idea is due to Hoare[14]). The vending machine is always willing to accept money, irrespective of whether it has the chocolates to uphold its pledge! However, it is at least honest – if it has no more chocolates to exchange for the money, it returns your coin. It is also a fairly clever vending machine; the slot in the machine allows only the correctly sized coin to be inserted, and other objects entered by dishonest or hungry customers are simply ignored. The first step in the statechart specification is to identify the states and the events in
the system. To the outside observer, there are two visible states: The \textit{READY} state in which the machine is willing to accept money, and the \textit{BUSY} state in which it processes a request for a chocolate. There are three relevant events: The event of a coin insertion into the machine, the event of a chocolate returned to the customer, and the event of a coin returned to the customer. This can be represented in a statechart as depicted in figure 4. Note the different types of arrows used to represent the different types of events.

One can now continue to refine the system C. Clearly, there must be some busy state in which C has chocolates to return to the customer, and some busy state in which no chocolates are available. Note in figure 5 how the enclosure of the states \textit{NO-CHOCS} and \textit{MORE-CHOCS} in the state \textit{BUSY} indicates that only one of those states can hold at any one time (that is, encapsulation enforces an XOR).

It now remains to specify the event(s) which lead to the states \textit{NO-CHOCS} and \textit{MORE-CHOCS} from the \textit{READY} state. Obviously, it is the same event in both cases, namely, the event \textit{coin-accepted}. Since the same event leads to all the states enclosed in the \textit{BUSY} state, we can use one arrow ending at the perimeter of the \textit{BUSY} state (see figure 6).

Suppose now that there is a vending machine with exactly the same functionality as C, but which dispenses toffees. Let us call this machine T. We can now design a vending machine called CT from the specifications of C and T, which operates as follows: There are two slots for entering coins, one for a toffee and one for a chocolate (the price is the same). Suppose that a customer wants to buy a toffee and enters a coin in that slot. As before, if there are any toffees available, the machine delivers a toffee. However, if there are no toffees available, CT first checks whether there are any chocolates available. If there are, it returns a chocolate instead of a toffee. If there are neither toffees nor chocolates available, it returns the customer's coin. A similar return policy applies if a customer requests a chocolate. Graphically CT can be specified by a statechart as depicted in figure 7. Note that the dashed line indicates the Cartesian product of the two machines, so that it is in principle possible to simultaneously enter a coin for a toffee and a coin for a chocolate. The dashed line thus enforces an AND condition for the states in the system.

Another useful modelling mechanism is the selection arrow. The selection arrow is indicated by a circled S on the arrowhead of an event arrow. It can be interpreted as a generic event, with a number of clearly defined values to indicate which state is triggered by the event. For example in a menu driven system, this mechanism can be used to indicate the event of selecting one of many possible menu items. Detailed examples using this mechanism are given in section 4.
Figure 5: Chocolate Vending Machine Stage 2

Figure 6: Chocolate Vending Machine
Figure 7: Chocolate and Toffee Vending Machine

The explanation above covers only those aspects of statecharts that we need to design and specify the GMA example in section 4. The reader is again urged to consult the original papers for more information.

3 The Graphical User Interface for GMA

The GMA (Graphical Modelling and Analysis) package is a performance modelling tool for telecommunications networks such as the South African public data network Saponet-P and the South African telephone network. The package has already been described in other papers ([3, 17]) and this section provides a summary of the overall project followed by more detail on the aspects relevant to this paper.

The GMA package runs under Unix, with graphics based on the X-Windows system using the X Toolkit and OSF/Motif. The structure of the GMA package is shown diagrammatically in figure 8. The package consists of three independent modular units: The Data Extraction Software, the Model Solution Kernel and the Graphical User Interface. Both the data extraction software and the model solution kernel are independent of the graphical interface; a standard interface to each module serves to hide its implementation details. This facilitates one of the main advantages of the package, namely its utility in modelling diverse computer network implementations. Such diverse modelling is achieved by modifying the front-end modules to cater for installation-specific features. The graphical interface controls the extraction of configuration and traffic data from the communication network under study. It presents a diagram of the network to the user and constructs the performance model of the network. The model is solved by the solution kernel, and results are presented graphically on the diagram.

One of the main goals of the GMA project was to produce a modelling package which could be used by network engineers with no training in the construction of statistical performance evaluation models. This implies that we had to design an interface that would present a graphical view of the network in a familiar form. This presented no problems in the case of data communication networks, as GMA could simply present the user with a geographical and/or a
schematic view of the network nodes. The user can then switch between the geographical and schematic view, or zoom into a specific node of the network (see figure 9).

The interface for the telephone network did however present immediate problems. Typical telephone networks are hierarchical in nature, with calls routed among the different levels, as depicted in figure 10. Moreover, for historical reasons the typical telephone network actually consists of various overlay networks, with the different networks connected by crossover points (see figure 11). In addition, the characteristics of the network differ depending on geographical location – for example, the trunking and routing decisions for metropolitan areas differ from that of rural areas. Telephone networks also contain many more nodes than the typical data communication network. For example, the South African telephone network contains approximately 1100 nodes, while Saponet-P contains 31 nodes.

A graphical interface for the modelling of a telephone network thus has to take all the above factors into account. The user should be presented with a hierarchical outline view, a view of only one level of the hierarchy, and localized geographical area views. A facility must also be available to zoom in on the small representation of a node, the small representation being necessitated by the large number of nodes in the network. It should be possible to switch between representation modes without affecting the network information known to the system.

A user-friendly interface for the telephone network modelling in GMA is therefore clearly a complex system. The design of such an interface has to take into account all the events possible in any one mode, and the effects of such events. We found a natural language specification of such an interface to be open to too much ambiguity, and started a search for a formal design and specification tool suited to our needs. Axiomatic (for example, [2]) and other textual specifications (such as BNF, see [15]) were discarded, because they lead to specifications that are difficult to read and maintain for large complex interfaces such as the GMA interface. We thus set out to find a graphical specification method whereby we could design and specify an event-driven interactive user interface. Specification systems without a formal underlying methodology (see [9]) were discarded, mainly because of our interest in the provable correctness of the GMA system ([19] discusses this issue). The need for a formal graphical specification method pointed to the use of some extension or variation of state transition diagrams, such as that of Wasserman [20], Zave[21] or Harel[10]. Wasserman's USE methodology uses ATN's (Augmented Transition
Figure 9: The Schematic and Geographical Modes in GMA
Figure 10: The Hierarchical Levels in a Telephone Network

Figure 11: Two Overlay Networks in a Telephone Network
Nets), and suffer from the state explosion problem. Although Zave’s sequence diagrams have the same expressive power as statecharts, we found the explicit expression on concurrency easier to understand and use. We thus set out to design and specify the GMA interface using statecharts.

4 The Design and Specification of the GMA Interface

In this section we show how the GMA interface was progressively designed and specified using statecharts. Throughout the discussion we point out how statecharts can be used to design all three the components of the user interface, and we comment on the ease of use of its concurrency specification mechanism. We shall follow a standard procedure in the design and specification of the GMA user interface. We always start with the design of the first component of the Seeheim model (i.e. the Presentation Component) and given the Presentation design, we proceed to specify the Dialogue Component in more detail. The Application Interface Component can then be designed using the information about the external events in the Dialogue Component.

The GMA package should present the user with a logo on startup time, after which the proper working screen should be displayed. So, a global view of the overall system can be specified as in figure 12 where there are two non-concurrent states: The LOGO state and the GMA-PROPER state. Note how the design for the actual presentation of the screens (the Seeheim Presentation Component) is visible from the statechart: the XOR of the two states LOGO and GMA-PROPER with the given events implies that those two windows/screens cannot be visible simultaneously. The default entry point to the system is the LOGO state, through the external event gma. The event ok places the system in the GMA-PROPER state, from which the event CTRL-Q will cause a quit from the system. Since there is no event back from GMA-PROPER to LOGO, it is not possible to switch between these two visual presentations. The ok event together with the LOGO and GMA-PROPER states define the Dialogue Component, while the external events gma and CTRL-Q define the Application-Interface Component.

Let us look at the state GMA-PROPER in more detail. The visual layout of the window must comply with the OSF/Motif style guide and our house rules on screen layout and as such consist of three display areas: A Status Display Area, a Menubar and a Working Area. All three these areas should be visible simultaneously. The window layout (Presentation Component) can for example be presented as in figure 13. The corresponding statechart, with the relevant events, is given in figure 14. Note that, although the design of the Presentation Component of the user interface is clearly only informal, the statecharts do allow the designer to form a preliminary picture of the layout of the window as the design progresses. As before, the design of the Application-Interface Component can be derived from the (only) external event ok.
Figure 13: The Main Window Layout for GMA

Figure 14: GMA-PROPER
Each of the three display areas can now be specified separately. The Status Display Area is the simplest of the three in terms of the possible events. It is an example of a display area where the user cannot enter any information but the application can set or display status information. The events that can be input to the Status Display Area are all of the same generic type (i.e. set one of the status flags) and we use the selection arrow to specify these events (see figure 15). Note that "illegal" events are ignored by the status bar, in that there is simply no reaction.

The second area of the GMA-PROPER window is the Menubar. The Menubar Area consists of a constantly visible menu bar, drop-down menus and pop-up selection boxes. Again, the selection of a menu item from the menu bar is specified using a selection arrow. Notice how an event can be aliased to hide implementation detail: The OptionSelected event can be implemented to allow multiple methods to select an option, such as a click on the mouse pointer and keyboard input (see figure 16). The MenuBar state itself is simple to specify, as illustrated in figure 17.

For the state DropdownMenus, we specify a generic dropdown menu to illustrate the applicability of statecharts to this menu type. Consider any drop-down menu system with n dropdown menus. Each menu is either visible (when it is selected) or not visible. However, more than one menu cannot be visible simultaneously. The user can move between menus by dragging the pointer to the left or the right, or by exiting a menu and selecting a new option (on the menu bar). The statechart that models state DropdownMenus is depicted in figure 18.

When a dropdown menu is visible, a number of options are displayed, with the pointer indicating the current (highlighted) option. One can move the pointer to switch options, and
then click to select a new option. Alternatively, one can use mnemonics or accelerators from the keyboard to choose any of the options in that particular drop-down menu. Figure 19 contains the statechart for the visible state of a drop-down menu.

As a typical example of a popup menu, we selected to design and specify a Threshold Editor box. In the GMA context objects are colour coded to indicate their 'safety' level according to a certain threshold. The system has default numerical values for all thresholds, but the user is allowed to change them with the Threshold Editor. The editing should occur graphically and/or numerically, and the changes in the values should be visible in colour and in numerical values. We use a scale with three different colours, where the boundaries of the coloured areas of the scale may be dragged to edit the thresholds. As usual in graphical user interface applications, three standard buttons are provided to commit the changes, cancel any changes, or to provide help. The Presentation Component of this popup menu is given in figure 20, with the corresponding statechart in figure 21.

The easiest specification in the Threshold Editor is the BUTTONS state. The mouse pointer may be in the buttons area of the Threshold Editor box, without being on any of the buttons. In that case, none of the buttons are selectable. We call this state the IDLE state. If the mouse pointer is on any of the buttons, then the buttons are potentially selectable. We call this state, for each button, its ACTIVE state. When the mouse pointer is clicked on a button, its corresponding reaction is activated. This is the SELECTED state for each button. So, the statechart looks like figure 22.

The statechart for the BUTTONS state in figure 22 is a clear example of how the Application-
Figure 19: A Visible Drop-down Menu

Figure 20: The Layout for the Threshold Editor

Figure 21: The Statechart for the Threshold Editor
Interface Component can be designed using statecharts. The external events from each of the button SELECTED states defines the (in X-Windows terminology) 'hook' between the interface and the application.

The NUMERALS and SCALE states can be specified in a straightforward fashion, reminiscent of the statechart for the Status Display Area in figure 15. The statecharts are given in figure 23.

Up to this point, we have illustrated how to design some standard graphical user interface features using statecharts. However, the third area of the GMA-PROPER state is the Working Area, which consists mostly of application-specific graphics as opposed to the general menu system. We will now show how the statecharts measured up to their potential in the design of the Working Area.

As explained in section 3, the Working Area must present to the user of the system both a geographical and a schematic view of the network. Additionally, the user must be able to zoom in on a specific geographical area, and switch between geographical and schematic views as required. Moreover the user should be able to select the hierarchy of the network currently visible (refer again to figure 10). The statechart to model these requirements are given in figure 24.

Each of the states in figure 24 can now be refined into a more detailed statechart. For example, let us consider the AREA-GEOGRAPHICAL state (see figure 25). Upon state entrance, a selection determines which geographical area should be displayed. The corresponding set of nodes are displayed (the NODE state indicates the graphical display of a node with its corre-
sponding links). For any of these nodes, the node can be active or selected. Note again that, as in the Threshold Editor Box, the system can be in an idle state. Further refinements can now be designed according to the action taken upon node selection.

The reader will notice that, in the more detailed statecharts, how the user interactions are modelled by the events among states. In the implementation of these statechart designs, these events usually specify the third component of the Seeheim model, namely, the Application-Interface Component.

This completes the detail of our design of the GMA direct-manipulation user interface with statecharts.

We believe that this section supports our claim on the superiority of statecharts as a specification mechanism for interactive event-driven systems.

5 Conclusion

In this paper we set out to show that statecharts are well-suited to the formal specification of event-driven user interfaces based on X. We believe that this claim is substantiated by the empirical results illustrated in section 4. In that section we discussed the design of a non-trivial direct-manipulation graphical user interface, and showed how the statecharts can be applied to design all three components of the Seeheim model. In summary, the main advantages of the statechart approach in our experience are the following:
Conclusion

• The statechart itself has been formalized completely, increasing the feasibility of its use in systems where formal correctness is an issue.

• Of the graphical design methodologies that we considered, the statecharts provide the easiest and most natural way to model concurrency. This is essential in the case of interfaces based on X-Windows.

• The statechart design can proceed hierarchical, ensuring that the designer can visually modularize the design so that the detail required at any stage is minimal.

• The statechart is a natural formalism for most programmers, so that the learning curve for the methodology is remarkably short.

• All the components of the Seeheim user interface model can be designed with the statecharts, providing a uniform approach to the design and specification of the user interface.

One of the only disadvantages of the statechart approach that we encountered, was that (specifically in object-orientated user interfaces) the state-based approach was not always the most natural approach to describe graphical objects. However, after a few examples we found that we could exploit the generic similarity between graphical object behaviour, so that most of our designs are adaptions of previous designs. This is in fact a hidden advantage, since one can thereby ensure that the behaviour of the interface is constant throughout the system.

References


5 Conclusion


PRODUCT FORM SOLUTIONS FOR MULTISERVER CENTRES WITH HIERARCHICAL CONCURRENCY CONSTRAINTS

AE Krzesinski
University of Stellenbosch
7600 Stellenbosch
South Africa

R Schassberger
Technische Universität Braunschweig
33 Braunschweig
Germany

July 1991

Abstract. The MultiServer centre with Hierarchical Concurrency Constraints (MSHCC) is an exponential multiserver with a queueing discipline which imposes restrictions on the number of customers of different types and subtypes that can be served simultaneously. The MSHCC centre is of interest because it represents a (not too exotic) quasi-reversible queue and can therefore be included in a product form network, and because it can be used to represent system features which could not previously be modelled using classical BCMP centres.


Keywords and phrases: blocking, concurrency constraints, product form solutions, queueing networks, queueing theory.

1 INTRODUCTION

This paper describes an extension of a queueing model first presented in [7]. Consider a service station serving customers of types \( c \) where \( c \in C \) and \( C \) is a countable set. Customers of type \( c \) arrive individually in independent Poisson streams with rate \( \lambda_c \). The customers, whether waiting or in service, form a queue in the order of their arrival. Each customer presents a demand for service time distributed exponentially with mean \( \mu \). The server gives service at certain positions of the queue. Upon entering service a customer is served to completion without interruption, whereupon the customer departs. When a service completion occurs, the corresponding customer departs, the gap in the queue is closed by the obvious shift, and the server scans the queue from the front searching for the first customer whose admission into service would not violate the following set of concurrency constraints. Firstly, maximally \( B \) customers can be served simultaneously where \( 1 \leq B \leq \infty \). The queue can thus be viewed as having \( B \) parallel servers. Next, the set of customer types \( C \) is partitioned as \( \{C_r; r \in R\} \) where \( R \) is a countable
set and maximally \( B_r \) customers whose types are in \( C_r \) are allowed to be in service simultaneously where \( r \in \mathcal{R} \) and \( 1 \leq B_r \leq \infty \). Finally, for each \( r \in \mathcal{R} \) the set \( C_r \) is partitioned as \( \{C_{rs}; s \in S_r\} \) where \( S_r \) is a countable set and maximally \( B_{rs} \) customers whose types are in \( C_{rs} \) are allowed to be in service simultaneously where \( s \in S_r \) and \( 1 \leq B_{rs} \leq \infty \). The \( C_{rs} \) can be further partitioned, with corresponding restrictions placed on the number of customers simultaneously in service, but we shall not go beyond the \( C_{rs} \) as this is a straightforward generalization of what we shall do. The queue discipline can thus be described as FCFS subject to concurrency constraints. We have therefore named the queue the MultiServer centre with Hierarchical Concurrency Constraints (MSHCC).

Our main results are (i) a product form expression for the steady state distribution of the queue; (ii) the statement that the queue is quasi-reversible [5] and can therefore be included in a network of quasi-reversible centres; (iii) efficient recursive expressions for computing the performance measures of a MSHCC centre. The original model introduced in [7, 8] did not partition the \( C_r \) and \( B_r = 1 \) for all \( r \). In [2] and [3] this was extended to allow \( B_r > 1 \). The hierarchical partitioning imposed by the MSHCC server is thus new. Finally, the model employs a state description more detailed than that used previously in the references quoted above — which greatly benefits the analysis.

The paper is organized as follows. Section 2 defines a more general queueing model. The global balance equations for this model are presented and are decomposed into two sets of partial balance equations. The first set of partial balance equations has a product form solution. It is shown that this solution also satisfies the second set of partial balance equations if certain conditions are imposed on the queueing discipline. Section 3 presents the MSHCC concurrency constraints. It is shown that the MSHCC queue satisfies these conditions. The MSHCC centre is next shown to be quasi-reversible [5]. Therefore a queueing network consisting of BCMP [1] and MSHCC centres has a product form solution. Section 4 uses aggregation techniques to reduce the joint probability distribution to a computationally tractable form. Section 5 derives the performance measures for the MSHCC centre. Finally, a numerical example is presented which illustrates how the concurrency constraints affect the performance of customers competing for service at a MSHCC centre.

2 THE MODEL

Consider a queue with state space \( S = \{0\} \cup_{n=1}^{\infty} C^n \) where 0 denotes the empty queue and \( C \) denotes a countable set of customer types. Let \( (c_n \ldots c_1) \) denote the state of a queue of length \( n \) where \( c_i \) denotes the type of the customer in position \( i \) in the queue where \( 1 \leq i \leq n \), \( n \in \mathbb{N} \) and \( c_i \in C \). For \( 1 \leq i \leq n \) define the functions

\[
1_{i,n}(c_n \ldots c_1) = \begin{cases} 
1 & \text{the customer in position } i \text{ is in service} \\
0 & \text{the customer in position } i \text{ is not in service}
\end{cases}
\]
Define
\[ k(c_n \ldots c_1) = \sum_{i=1}^{n} 1_{i,n}(c_n \ldots c_1) \leq 1 \]
so that \( k(c_n \ldots c_1) \) is the number of customers being served in \((c_n \ldots c_1)\).

Let customers of type \( c \) arrive to the queue individually in independent Poisson streams with rate \( \lambda_c \) such that \( \sum_{c \in \mathcal{C}} \lambda_c < \infty \). Let arriving customers join the tail of the queue and let the front of the queue be identified with position 1. Let service on each position served be non-interruptible and take place at the positive rate \( \mu_n \) where \( n \) is the length of the queue. When service terminates at position \( i \), let the ensuing state be \((c_n \ldots c_{i+1}c_{i-1} \ldots c_1)\).

For the corresponding Markov process, which is obviously regular, the global balance equations are given by
\[ P(0) \sum_{c \in \mathcal{C}} \lambda_c = \sum_{c \in \mathcal{C}} P(c)\mu_1 \]
and
\[ P(c_n \ldots c_1) \left( \sum_{c \in \mathcal{C}} \lambda_c + k(c_n \ldots c_1)\mu_n \right) = \lambda_{c_n} P(c_{n-1} \ldots c_1) + \sum_{c \in \mathcal{C}} \sum_{i=0}^{n} P(c_n \ldots c_{i+1}c_i \ldots c_1)1_{i+1,n+1}(c_n \ldots c_{i+1}c_i \ldots c_1)\mu_{n+1} \]
(1)
where the term \( \lambda_{c_n} P(c_{n-1} \ldots c_1) \) is replaced by \( \lambda_{c_1} \) if \( n = 1 \).

We wish to determine solutions for equation (1) which also satisfy the partial balance equations
\[ P(c_n \ldots c_1)k(c_n \ldots c_1)\mu_n = \lambda_{c_n} P(c_{n-1} \ldots c_1) \]
(2)
Equation (2) immediately yields
\[ P(c_n \ldots c_1) = P(0) \prod_{i=1}^{n} \frac{\lambda_{c_i}}{\mu_i k(c_i \ldots c_1)} \]
(3)
In order for equation (3) to satisfy equation (1) for any choice of the \( \lambda_c \) it is necessary and sufficient that equation (4) satisfies the partial balance equations
\[ P(c_n \ldots c_1)\lambda_c = \sum_{i=0}^{n} P(c_n \ldots c_{i+1}c_i \ldots c_1)1_{i+1,n+1}(c_n \ldots c_{i+1}c_i \ldots c_1)\mu_{n+1} \]
(4)
or equivalently (by substituting equation (3) into equation (4)) that
\[ 1 = \sum_{i=0}^{n} \frac{k(c_n \ldots c_1) \ldots k(c_{i+1} \ldots c_1)}{k(c_n \ldots c_{i+1}c_i \ldots c_1) \ldots k(c_{i} \ldots c_1)} \]
(5)
In order to find sufficient conditions for equation (5) to be satisfied, assume
2 THE MODEL

Assumption 1 For all \((c_n \ldots c_1)\) and for all \(1 \leq i \leq n\)

\[
l_{i,n}(c_n \ldots c_1) = l_{i,i}(c_i \ldots c_1)
\]

Assumption 1 thus implies that the composition of the queue up to position \(i\) determines whether the customer at position \(i\) is admitted into service. Applying assumption 1 to equation (5) yields

\[
1 = \sum_{i=0}^{n} \frac{k(c_n \ldots c_1) \ldots k(c_{i+1} \ldots c_1)}{k(c_n \ldots c_{i+1}cc_i \ldots c_1) \ldots k(cc_i \ldots c_1)} l_{i+1,i+1}(cc_i \ldots c_1)
\]

Let the right hand side of equation (6) be denoted as \(V_c(c_n \ldots c_1)\) if \(n > 0\), and as \(V_c\) if \(n = 0\). Note that \(V_c = 1\). Next define

\[
\nu_c(c_n \ldots c_1) = \max(i : l_{i+1,n+1}(c_n \ldots c_{i+1}cc_i \ldots c_1) = 1, 0 \leq i \leq n)
\]

and henceforth, for fixed \(c\) and fixed \((c_n \ldots c_1)\), write \(\nu_c(c_n \ldots c_1) = \nu\). Next assume that for all \(c\)

Assumption 2 If \(\nu < n\) then for all \(0 \leq i \leq \nu < j \leq n\)

\[
k(c_j \ldots c_{i+1}cc_i \ldots c_1) = k(c_j \ldots c_1)
\]

Assumption 3 If \(\nu = n\) then for all \(0 \leq i \leq n\)

\[
k(c_n \ldots c_{i+1}cc_i \ldots c_1) = k(c_n \ldots c_1) + 1
\]

Assumptions 2 and 3 will later be shown to be satisfied when the system of hierarchical constraints described in section 3 is used to regulate the admission of customers into service at the MSHCC centre.

If \(\nu = n\) then application of assumption 3 yields

\[
V_c(c_{n} \ldots c_1) = \frac{1 + k(c_n \ldots c_1)V_c(c_{n-1} \ldots c_1)}{1 + k(c_n \ldots c_1)}
\]

and if \(\nu < n\) application of assumption 2 yields

\[
V_c(c_n \ldots c_1) = V_c(c_\nu \ldots c_1)
\]

THEOREM 2.1 Under the assumptions 1, 2 and 3 the Markov process describing the fluctuation of the queue has its global balance equations solved by equation (3), and equation (3) also satisfies the partial balance equations (2) and (4).
3 CONCURRENCY CONSTRAINTS

PROOF. Given $V_c = 1$ and assuming that it has already been shown that $V_c(c_k \ldots c_1) = 1$ for all $c$ and $(c_k \ldots c_1)$, then the proof of the theorem follows immediately by induction. 

We now show that the queueing process is quasi-reversible by demonstrating that for all $c \in C$ (i) the arrival times of type $c$ customers form independent Poisson processes and (ii) the departure times of type $c$ customers form independent Poisson processes. Both of these conditions hold: (i) is part of the definition of the model and (ii) is a consequence of equation (4).

3 CONCURRENCY CONSTRAINTS

Consider the queue described in section 2. Let $x = (c_n \ldots c_1)$ denote the state of the queue and let $k(x), k_r(x)$ and $k_{rs}(x)$ denote the total number of customers, the total number of $C_r$- and the total number of $C_{rs}$- customers served in state $x$ respectively. The MSHCC concurrency constraints require that

$$k(x) = \sum_{i=1}^{n} 1_{i,n}(c_n \ldots c_1) \leq B$$
$$k_r(x) = \sum_{i=1}^{n} 1_{i,n}(c_n \ldots c_1) 1(c_i \in C_r) \leq B_r$$
$$k_{rs}(x) = \sum_{i=1}^{n} 1_{i,n}(c_n \ldots c_1) 1(c_i \in C_{rs}) \leq B_{rs} \tag{8}$$

where $1(E)$ is the indicator function of the event $E$.

Assumption 4 The server, upon completing the service of a customer, searches the queue from the front (queue position 1) to the end for the first customer whose admission into service would not violate any of the MSHCC concurrency constraints.

THEOREM 3.1 Under the assumptions 1, 2, 3 and 4 the MSHCC queue is a special case of the queue described in section 2 and satisfies the conditions of theorem 2.1.

PROOF. The MSHCC queue is clearly a special case of the queue presented in section 2, with the functions $i_{i,n}(\cdot)$ all well defined in terms of the concurrency bounds $B$, $B_r$ and $B_{rs}$, and assumption 4 defining how the server searches the queue to select a customer to take into service. Given that arrivals join the end of the queue and that service is non-interruptible then assumption 4 clearly implies assumption 1. It remains to show that assumptions 2 and 3 hold.

As in theorem 2.1, for a fixed queue configuration $(c_n \ldots c_1)$, and for a fixed customer type $c$, let $\nu_c(c_n \ldots c_1) = \nu$. Clearly assumption 3 is satisfied. As for
assumption 2, observe that for $0 \leq i \leq \nu < j \leq n$

$$k(c_j \ldots c_{\nu+1} c_{\nu} \ldots c_1) = k(c_{\nu} \ldots c_1) + \sum_{k=\nu}^{j} 1_{k+1,j+1}(c_j \ldots c_{\nu+1} c_{\nu} \ldots c_1)$$

$$= k(c_{\nu} \ldots c_1) + \sum_{k=\nu}^{j} 1_{k+1,j+1}(c_j \ldots c_{\nu+1} c_{\nu} \ldots c_{i+1} c_i \ldots c_1)$$

It is obvious that for $0 \leq i \leq \nu < n$

$$k(c_{\nu} \ldots c_{i+1} c_i \ldots c_1) = k(c_{\nu} \ldots c_1)$$

so that

$$k(c_j \ldots c_{\nu+1} c_{\nu} \ldots c_1) = k(c_j \ldots c_{\nu+1} c_{\nu} \ldots c_1)$$

It therefore remains to show that for $\nu < j \leq n$

$$k(c_j \ldots c_{\nu+1} c_{\nu} \ldots c_1) = k(c_j \ldots c_1) \quad (9)$$

Let $x = (c_j \ldots c_{\nu+1} c_{\nu} \ldots c_1)$ and label the customers in positions $\nu + 1$ and $\nu + 2$ as the tag-1 and tag-2 customers respectively. Let $x'$ result from $x$ by exchanging the tagged customers, and let $y$ result from $x'$ by removing the tag-1 customer and shifting forward to close the gap. Thus $y = k(c_j \ldots c_1)$. We intend to demonstrate that $k(x) = k(x') = k(y)$ which will prove equation (9) to be correct.

Assume that $c \in C_r$. First consider the case where $c_{\nu+1} \in C_r$. Then either $k(x_0) = B$ or $k_r(x_0) = B_r$ or $k_{rs}(x_0) = B_{rs}$ where $x_0 = (c_{\nu} \ldots c_1)$ and “or” means “inclusive or”. The status $S(erved)$ or $W(ait)$ of the tagged customers changes as $x$ becomes $x'$ and the status of the untagged customers remains unaltered as $x$ becomes $x'$ becomes $y$. Thus $k(x) = k(x') = k(y)$ which proves equation (9).

Next consider the case where $c_{\nu+1} \in C_r$ for some $t \neq s$. Then either $k(x_0) = B$ or $k_r(x_0) = B_r$. Also $k_{rt}(x_0) < B_{rt}$ otherwise the tag-2 customer could not be served after $x$ becomes $x'$ and hence the tag-1 customer would be served after the exchange, contradicting the definition of $\nu$. Thus the status of the tagged customers changes as $x$ becomes $x'$ and the status of the untagged customers remains unaltered as $x$ becomes $x'$ becomes $y$. Thus $k(x) = k(x') = k(y)$ which proves equation (9).

Finally consider the case where $c_{\nu+1} \notin C_r$. Then $k(x_0) = B$. The tag-2 customer changes its status from $W$ to $S$ as $x$ becomes $x'$ otherwise the tag-1 customer would still be served after the exchange. Thus equation (9) follows by the previous argument which completes the theorem.

A closer look at the above theorem reveals an unforeseen fact, namely that the total number of customers served in the queue remains unchanged under a permutation of the customers in the queue. This is expressed in

**COROLLARY 3.2** For the MSHCC queue

$$k(c_n \ldots c_1) = k(c_{\sigma(n)} \ldots c_{\sigma(1)}) \quad (10)$$

where $\sigma$ denotes any permutation of $(1 \ldots n)$ and $(c_n \ldots c_1)$ is any state of $S$. 

74
4 AGGREGATION

PROOF. It suffices to show that for \(0 \leq 1 < n\) and for all \((c_n \ldots c_1)\)

\[ k(c_n \ldots c_1) = k(c_{i-1} \ldots c_{i+1} c_i \ldots c_1) \] (11)

Let \(\nu = \nu_c(c_{n-1} \ldots c_1)\) be defined as in section 2. If \(\nu = n - 1\) then equation (11) follows directly from assumption 3. If \(\nu =< n - 1\) then equation (11) follows directly from assumption 2 for all \(0 \leq i \leq \nu\).

For \(i = \nu + 1\) write \(x = (c_{n-1} \ldots c_{\nu+1} c_n c_{\nu} \ldots c_1)\) and \(x' = (c_{n-1} \ldots c_{\nu+2} c_n c_{\nu+1} \ldots c_1)\). As shown in theorem 3.1, \(k(x) = k(x')\) which proves equation (11) for \(i = \nu + 1\). Starting from \(x'\) exchange the customers in positions \(\nu + 2\) and \(\nu + 3\). This leaves the status (W or S) of all customers unchanged, which proves equation (11) for \(i = \nu + 2\). Further such exchanges proves equation (11) for the remaining i. \(\Box\)

4 AGGREGATION

The steady state distribution \(P(c_n \ldots c_1)\) which presents the MSHCC centre in terms of the order of the customers is too detailed to be of practical use when computing the performance measures of the MSHCC centre. Equation (10) is the key for obtaining results concerning aggregated states. Define

\[ S_{\bar{m}} = \{ x \in S : m_c(x) = m_c, c \in C \} \]

where \(m_c(x)\) is the number of type \(c\) customers present when the queue is in state \(x\), and where \(\bar{m} = (m_c)_{c \in C}\) where \(m_c \geq 0\). Let \(\bar{m} - c\) be obtained from \(\bar{m}\) by substituting \(m_c - 1\) for \(m_c\), this being defined only for \(m_c > 0\).

Corollary 3.2 allows us to introduce an abuse of notation for the MSHCC queue, namely for \(x \in S_{\bar{m}}\) to write \(k(\bar{m}) = k(x)\) to denote the total number of customers served in a state from \(S_{\bar{m}}\).

THEOREM 4.1 For the MSHCC queue

\[ k(\bar{m} + c) = \begin{cases} k(\bar{m}) & |\bar{m}_r| \geq B_r \\ k(\bar{m}) & |\bar{m}_r| < B_r \text{ and } m_r \geq B_r \\ k(\bar{m}) + 1 & |\bar{m}_r| < B_r \text{ and } m_r < B_r \end{cases} \] (12)

where \(c \in C_r\).

PROOF. This is a direct consequence of the MSHCC service discipline. \(\Box\)

THEOREM 4.2 For the MSHCC queue

\[ k(\bar{m})P(S_{\bar{m}}) = \sum_{c \in C, m_c > 0} \rho_c P(S_{\bar{m} - c}) \] (13)

where \(\rho_c = \lambda_c / \mu|\bar{m}|\)
5 MSHCC PERFORMANCE MEASURES

PROOF. The proof is given in [6].

Equation (13) permits the recursive calculation of $P(S_{\bar{m}})$. However, simpler recursions are available in the domain where $k(\bar{m}) < B$. These recursions, which are presented in section 5, require the following lemma, where $m_r(x)$ and $m_{rs}(x)$ denote the number of $C_r$- and $C_{rs}$- customers present in state $x$, where $x \in S$.

**Lemma 4.1** For $x \in S$ such that $k(x) < B$

$$
k_r(x) = \begin{cases} B_r & m_r(x) \geq B_r \\ \sum_{s \in S_r} m_{rs}(x) \land B_{rs} & m_r(x) < B_r \end{cases}$$

(14)

where $p \land q$ is the smaller of the two integers $p$ and $q$. Furthermore, if $k(x) < B$ and $m_r(x) < B_r$, then $k_{rs}(x) = m_{rs}(x) \land B_{rs}$ for all $s$ in $S_r$.

**Proof.** This is a direct consequence of the MSHCC service discipline.

Lemma 4.1 implies that for $\bar{m}$ such that $k(\bar{m}) < B$ we can allow a further abuse of notation for the MSHCC queue, namely for $x \in S_{\bar{m}}$ to write $k_r(\bar{m}) = k_r(x)$ to denote the number of $C_r$- customers served in a state from $S_{\bar{m}}$. If in addition, $\bar{m}$ is such that $m_r(x) < B_r$ for $x \in S_{\bar{m}}$ then we can similarly write $k_{rs}(\bar{m}) = k_r(x)$ to denote the number of $C_{rs}$- customers served in a state from $S_{\bar{m}}$.

**Lemma 4.3** For $\bar{m}$ such that $k(\bar{m}) < B$

$$
k_r(\bar{m})P(S_{\bar{m}}) = \sum_{c \in C_r, m_c > 0} \rho_c P(S_{\bar{m}-c})$$

(15)

for all $r$. Additionally, if $m_r(x) < B_r$ for $x \in S_{\bar{m}}$, then for this $r$ and for all $s \in S_r$

$$
k_{rs}(\bar{m})P(S_{\bar{m}}) = \sum_{c \in C_{rs}, m_c > 0} \rho_c P(S_{\bar{m}-c})$$

(16)

**Proof.** The proof is given in [6].

5 MSHCC PERFORMANCE MEASURES

This section presents efficient recursions to compute the performance measures at a MSHCC centre. The recursions employ equation (12) and are confined to the domain $1 \leq k(\bar{m}) < B$ so that equations (14) and (15) can be exploited. The definitions and theorems in this section therefore all assume that $1 \leq k(\bar{m}) < B$.

Recall $\bar{m} = (m_c)_{c \in C}$ and define $\bar{m} - c = (m_c)_{c \in C_r}$ where $m_c \geq 0$. Define $|\bar{m}| = n$. Let $\bar{m} - c$ be obtained from $\bar{m}$ by substituting $m_c - 1$ for $m$, this being defined
only for $m_c > 0$. Let $M_c$ denote a random variable distributed as the number of type $c$ customers present in the queue at equilibrium. Define a random variable $\bar{M}_r = (M_c)_{c \in S_r}$. Define

\[ P^\bar{m}_b(r + 1, n) = \Pr(k(\bar{m}) = b, \bar{M}_r = \bar{m}_r, |\bar{M}_i| = 0 \text{ for } i > r) \]

Define $\bar{B}_r = (B_{rs})_{s \in S_r}$ and denote $\bar{m}_r < \bar{B}_r$ if $m_{rs} < B_{rs}$ for all $s \in S_r$ and $\bar{m}_r \geq \bar{B}_r$ otherwise. Let $\rho_c = \lambda_c / \mu_r$.

**Theorem 5.1** For $1 \leq r < R$ and $\bar{m}_r < \bar{B}_r$,

\[ |\bar{m}_r| P^\bar{m}_b(r + 1, n) = \sum_{c \in C_r, m_c > 0} \rho_c P^{\bar{m}_r - c}_{b-1}(r + 1, n - 1) \]

**Theorem 5.2** For $1 \leq r < R$ and $\bar{m}_r \geq \bar{B}_r$ and $|\bar{m}_r| < B_r$,

\[ k_r(\bar{m}) P^\bar{m}_b(r + 1, n) = \sum_{s \in S_r} \sum_{m_{rs} = B_{rs}} \rho_c P^{\bar{m}_r - c}_{b-1}(r + 1, n - 1) \]

\[ + \sum_{s \in S_r} \sum_{m_{rs} \leq B_{rs}, m_{rs} - c < \bar{B}_r} \rho_c P^{\bar{m}_r - c}_{b-1}(r + 1, n - 1) \]

\[ + \sum_{s \in S_r} \sum_{m_{rs} > B_{rs}, m_{rs} - c \geq \bar{B}_r} \rho_c P^{\bar{m}_r - c}_{b-1}(r + 1, n - 1) \]

**Theorem 5.3** For $1 \leq r < R$ and $|\bar{m}_r| \geq B_r$,

\[ B_r P^\bar{m}_b(r + 1, n) = \sum_{c \in C_r, m_c > 0} \rho_c P^{\bar{m}_r - c}_{b-1}(r + 1, n - 1) \]

\[ + \sum_{s \in S_r} \sum_{m_{rs} > B_{rs}, |\bar{m}_r - c| < B_r} \rho_c P^{\bar{m}_r - c}_{b-1}(r + 1, n - 1) \]

\[ + \sum_{s \in S_r} \sum_{m_{rs} \leq B_{rs}, |\bar{m}_r - c| < B_r} \rho_c P^{\bar{m}_r - c}_{b-1}(r + 1, n - 1) \]

For $1 \leq r < R$ define

\[ P_b(r + 1, n) = \Pr(k(\bar{m}) = b, |\bar{M}_i| = 0 \text{ for } i > r) \]

Clearly

\[ P_b(r + 1, n) = P_b(r, n) + \sum_{m_r, |\bar{m}_r| > 0} P^\bar{m}_b(r + 1, n) \] (17)
where the $P_b^{\tilde{m}_R}(\tau+1, n)$ are given in theorems 5.1, 5.2 and 5.3 above. These theorems thus yield a recursive procedure to compute $P_b(R, n)$.

Next define

$$P_b^{\tilde{m}_R}(n) = \Pr(k(\tilde{m}) = b, \tilde{M}_R = \tilde{m}_R)$$

**THEOREM 5.4** For $\tilde{m}_R < \tilde{B}_R$

$$|\tilde{m}_R| P_b^{\tilde{m}_R}(n) = \sum_{c \in \mathcal{C}_{\tilde{R}, m_c > 0}} \rho_c P_b^{\tilde{m}_R-c}(n-1)$$

**THEOREM 5.5** For $\tilde{m}_R \geq \tilde{B}_R$ and $|\tilde{m}_R| < B_R$

$$k_R(\tilde{m}) P_b^{\tilde{m}_R}(n) = \sum_{s \in \mathcal{S}_R} \sum_{c \in \mathcal{C}_{R_s, m_c > 0}} \rho_c P_b^{\tilde{m}_R-c}(n-1)$$

$$+ \sum_{m_{R_s} = B_{R_s}} \sum_{c \in \mathcal{C}_{R_s, m_c > 0}} \rho_c P_b^{\tilde{m}_R-c}(n-1)$$

$$+ \sum_{m_{R_s} \leq B_{R_s}} \sum_{c \in \mathcal{C}_{R_s, m_c > 0}} \rho_c P_b^{\tilde{m}_R-c}(n-1)$$

$$+ \sum_{m_{R_s} > B_{R_s}} \sum_{c \in \mathcal{C}_{R_s, m_c > 0}} \rho_c P_b^{\tilde{m}_R-c}(n-1)$$

**THEOREM 5.6** For $|\tilde{m}_R| \geq B_R$

$$B_R P_b^{\tilde{m}_R}(n) = \sum_{c \in \mathcal{C}_{R, m_c > 0}} \rho_c P_b^{\tilde{m}_R-c}(n-1)$$

$$+ \sum_{m_{R_s} = B_{R_s}} \sum_{c \in \mathcal{C}_{R_s, m_c > 0}} \rho_c P_b^{\tilde{m}_R-c}(n-1)$$

$$+ \sum_{m_{R_s} > B_{R_s}} \sum_{c \in \mathcal{C}_{R_s, m_c > 0}} \rho_c P_b^{\tilde{m}_R-c}(n-1)$$

Next define

$$P_b(n) = \Pr(k(\tilde{m}) = b)$$

Clearly

$$P_b(n) = P_b(R, n) + \sum_{\tilde{m}_R | \tilde{m}_R > 0} P_b^{\tilde{m}_R}(n)$$

78
where the \( P_5^{\bar{m}_R}(n) \) are given in theorems 5.4, 5.5 and 5.6 above and \( P_5(R, n) \) is given in equation (17). We thus have a recursive procedure to compute \( P_5(n) \) for \( 1 \leq b < B \).

Let \( L_b(n) \) denote the average number of type \( C_R \) customers at the MSHCC centre when \( k(\bar{m}) = b \). Then for \( 1 \leq b < B \)

\[
L_b(n) = \sum_{\bar{m}_R} |\bar{m}_R| P_5^{\bar{m}_R}(n)
\]

Let \( L(n) \) denote the average number of type \( C_R \) customers at the MSHCC centre.

**THEOREM 5.7**

\[
BL(\lambda) = \sum_{b=1}^{B-1} (B - b)L_b(n) + \rho L(n - 1) + \rho_R
\]

where \( \rho_R = \sum_{c \in C_R} \rho_c \) and \( \rho = \sum_{c \in C} \rho_c \).

The proofs of theorems 5.1 through 5.7 are given in [3].

6 APPLICATIONS

MULTIPORTED MEMORY

Consider [7, 8] a computer system consisting of \( N \) processors accessing \( K \) memory modules via a partitioned multiple bus system. Each of the \( G \) groups of \( B \) buses gives access to a subset of \( K/G \) memory modules. Each memory module \( k \) is \( n_k \)-ported so that maximally \( n_k \) processors can access memory module \( k \) simultaneously. The system is modelled as a queueing network consisting of an IS centre representing the processors and \( G \) MSHCC centres, each representing one group of buses and the associated memory modules. Each MSHCC centre consists of \( B \) servers which represent the \( B \) buses in its group. The customers in the network belong to \( K \) classes.

A processor service interval followed by a data transfer to/from memory module \( k \) is modelled as a customer departing from the processor service centre, and moving to the \( g^\text{th} \) MSHCC centre where group \( g \) contains an access path to memory module \( k \). The customer changes class to class \( k \) and queues for service at the MSHCC centre. The class \( k \) customer enters into service if one of the \( B \) servers is free (a bus is available) and if at most \( n_k \) class \( k \) customers are in service (at most \( n_k \) processors can access memory module \( k \) simultaneously). The fact that different classes of customers are permitted allows each processor to preferentially access a different set of memory modules and to issue memory access requests at a rate differing from the request rates of the other processors.
A MESSAGING CARD

In some distributed architectures such as telephone switching exchanges, the messaging function between a high level peripheral (decentralized call processor) and a lower level one (line or trunk controller) is performed on the high level side by a specialized processor (the messaging card) which controls simplex channels to the lower level peripherals. The processor time is partitioned into $B$ fixed time slots, each of which is allocated to a process whose function is to send outgoing messages. The exchange forwards messages to $K$ destinations, each of which is reachable on any of $n_k$ channels. When a buffer is queued for transmission it has to wait for a process to be available to service the request and for an outgoing channel to be free. The messaging card can be modelled by a MSHCC centre consisting of $B$ servers (the transmission processes) serving customers belonging to $K$ classes, each with its own concurrency limit $n_k$.

MULTILAYER WINDOW FLOW CONTROL

In X.25 packet switched networks window flow control is implemented at two levels. At the data link level maximally $B$ frames may be unacknowledged between two communicating nodes; at the network level maximally $n_k$ packets may be unacknowledged on each of $K$ virtual circuits multiplexed onto the data link.

A MSHCC centre can be used to model the above window flow control policy as follows [4]. The customers arriving at the MSHCC centre belong to $K$ groups. A customer of group $k$ represents a packet belonging to virtual circuit $k$. Packets are queued for transmission in the order of their arrival. A customer of group $k$ will be admitted into service (transmitted on the link) provided that fewer than $B$ servers are occupied and maximally $n_k - 1$ other group $k$ customers are in service. A customer occupies a server for a time corresponding to the transmission delay for a packet and the time taken for an acknowledgement to be returned to the node. Thus maximally $B$ packets may be in transit at the data link level and maximally $n_k$ at the network level for each virtual circuit $k$.

MULTISERVER MULTIQUEUE SYSTEMS

A Multiserver MultiQueue system (MSMQ) consists of a set of $N$ queues and $S$ servers. Customers arrive to the queues individually in independent Poisson streams. The queues have infinite capacities and the queueing discipline within each queue is FCFS. Servers move cyclically from one queue to another and every time a server visits a queue at most one waiting customer is served. The movement of servers from queue to queue takes place in zero time. Two server utilization policies are possible: the $1 \times Q$ policy states that only 1 server can be serving a queue at a given time and the $S \times Q$ policy permits all $S$ servers to simultaneously attend to one queue.
The dynamics of an MSMQ system operating according to the $1 \times Q$ policy is remarkably similar to the MSHCC queue with $S$ servers and $N$ customer types. The customer types can be identified with the MSMQ queues. In each case maximally one customer per type or queue can be in service simultaneously. The only difference is in the polling order: in the case of MSHCC it is FCFS subject to the concurrency constraints, and in the MSMQ system it is cyclical. However, if the scheduling algorithm minimizes the number of empty queues, the impact of the polling order on the average customer delay can be expected to be minimal.

The similarity between the MSHCC and MSMQ systems suggests that the MSHCC queue can provide a very accurate approximation of the $1 \times Q$ MSMQ system. Recent simulation studies [9] confirm that the accuracy of the MSHCC approximation to the $1 \times Q$ system usually lies within the confidence interval of the simulation process.

References


A Reusable Kernel For the Development of Control Software

W. Fouché and P.J.A. de Villiers
Institute for Applied Computer Science
University of Stellenbosch, Stellenbosch 7600

Abstract
A kernel has been developed which can be used to simplify the development of control software. It can be used to control several physical machines which are interconnected via a fast local area network. One or more virtual machines are simulated on each physical machine. The kernel was used to implement an operating system consisting of a file server, a name server and a shell. However, it can also be used to support control software for embedded systems. The performance of the kernel has been measured and found to be comparable to other kernels.

1 Introduction
The long term goal of the HYBRID project is to develop control software for distributed hardware. One method which simplifies this task is to develop reusable software modules. In this respect a programming language which directly supports modules is invaluable. We chose to use Modula-2[20]. The programming of bare hardware is a time consuming but necessary part of the development of most control systems. Benefits can be obtained by packaging such low-level code as a reusable kernel so that a designer can use it without necessarily understanding its implementation details. The design of electronic systems was simplified significantly when engineers packaged reusable designs as integrated circuits. Any integrated circuit can be used as a “black box” as long as its function is understood properly, its internal details being irrelevant. We believe that an efficient and reliable kernel can do the same for the design of control software. The design and implementation of a reusable, reliable and efficient kernel was therefore the major initial activity of this project.

2 A Kernel Supporting Virtual Machines
The HYBRID kernel provides the basic functionality needed to implement control systems: support for multiple processes, memory management, interprocess communication and operations on peripheral devices. To simplify a control system it is necessary to decompose it into simpler subsystems. An efficient communication facility is therefore needed to enable these subsystems to cooperate. Communication must be based on simple but general principles in order to avoid subtle
errors. CSP[11] presents a theory which can be used as a basis for the orderly design of concurrent systems. Concurrent processes are only allowed to communicate by exchanging messages according to simple well-defined rules. The success of the transputer provides evidence that CSP can be used as a basis for the design of practical systems. Although transputer networks can host control software efficiently, it is restrictive that the number of transputers in a given network is fixed.

The HYBRID kernel transforms different physical machines, interconnected by a high performance local area network, into several communicating virtual machines (VMs) which are similar in many ways to transputers. VMs execute processes which communicate according to the principles of CSP. Since VMs can be created dynamically the HYBRID kernel relaxes a constraint of transputer networks—a fixed number of transputers.

The kernel makes simple and elegant designs feasible because a virtual machine can be dedicated to each autonomous task. If a task does not need the power of a dedicated physical machine, more than one VM can be executed by the same physical machine. It is a matter of separating concerns: we believe that using the concept of VMs, each dedicated to a specific task, can lead to elegant designs, while multiplexing a physical machine among several VMs is a separate matter which is handled by the kernel. A designer can thus ignore the underlying hardware and think in terms of communicating VMs.

3 Design of the Kernel

A detailed discussion of the internal structure of the HYBRID kernel in which the various design decisions are motivated is presented in [9]. Here we present only an overview of the most important concepts. The design of the kernel is conservative rather than innovative, exploiting successful concepts found in existing systems such as Amoeba[14, 19, 15], Chorus[10, 18, 1], Mach[17, 16] and V[6, 4, 5]. These systems implement similar services by using different techniques as dictated by the experience and beliefs of their designers.

In order to enable a system designer to think in terms of abstract concepts alone, the kernel hides the properties of the underlying physical machine. Another important feature of the kernel is scalability—if better performance is needed another processor can be added without changing the logical design of the control system which is designed in terms of virtual machines. A VM is an abstract machine with the following properties:

- It has a separate protected address space which contains code and data.
- A VM can execute one or more (light-weight) processes which share the same address space.
- Processes communicate by sending and receiving messages.
Figure 1: Multiple VMs Supported by a Single Physical Machine

Figure 1 shows a single physical machine supporting several VMs. To maintain system integrity the kernel prevents one VM from accessing the address space of another. This makes it impossible for any VM to accidentally (or maliciously) modify the private memory area of another. Hardware support for memory protection is thus essential.

A process is represented in the kernel by its process descriptor which is used to store its current state. Process descriptors of processes which are ready to run are linked to form a process queue. In a similar way a VM is represented by its VM descriptor. Each VM descriptor contains a pointer to its associated process queue. VM descriptors of VMs which can be enabled are linked to form a VM queue. The scheduler first selects a VM to activate and then selects an appropriate process to run. VMs are time-sliced and may have different priorities, but a process is allowed to run until an interprocess communication (IPC) operation is invoked or until the time-slice of its VM expires.

Unbuffered synchronous message passing is used to exchange information between processes via ports. Ports are global identifiers used to route messages to their destinations. We assume most messages to be short. Bershad et al.[3] found that nearly 80% of all messages transmitted in Topaz, the operating system of the Firefly multiprocessor workstation developed at DEC SRC, are more or less 32 bytes in length. Thus it was decided to use copying to transfer messages between processes on the same machine. Alternatively, page remapping techniques[8] can be used, but the extra complexity seems to be worthwhile only for large messages. Three IPC operations are defined: Transaction, ReceiveRequest and SendReply. Transaction is used to send a request message to a process and to receive the associated reply message. ReceiveRequest receives a message and SendReply returns a reply message to a process. A sender process, making use of the Transaction
primitive to send a message, remains suspended until the request message has been accepted and a reply message returned. Similar models for interprocess communication have been found to work well in the kernels of the Amoeba[15] and V[5] systems.

All device servers (peripheral drivers) are coded in a similar way and it is necessary to understand just a few basic principles in order to add a server for a new peripheral device. A device server receives requests which are entered in a request queue. While requests are available, one is selected from this queue and the appropriate peripheral operation is started. The device server is then deactivated by calling the scheduler to resume the next ready process. The interrupt which signals completion of the peripheral operation reactivates the device server in order to start the next peripheral operation.

It can thus be seen that interrupt handling, process scheduling and message passing are closely interconnected although this is of no concern to a programmer at the user level—a VM is a clearly defined deterministic machine which supports a natural execution environment for one or more cooperating sequential processes. The IPC facility provides a simple means of process synchronisation.

Although the development of the HYBRID kernel was motivated primarily by a requirement to implement an operating system for distributed hardware, we later realised that with minor modifications the kernel could also support other classes of control software.

4 A VM-based Operating System

The operating system which was developed as a first application of the kernel is used for research and educational purposes. Since an operating system is an example of a complex control system this application represents a reasonable test of the kernel. The system is based on the client-server concept and was deliberately kept as simple as possible, its various services being made accessible via the standard communication facilities offered by the kernel. Efficient servers can be designed as a group of cooperating processes which are supported naturally by a VM. Clients request servers to perform operations on their behalf by transmitting messages. Servers send reply messages to inform clients of the outcome of requested operations or to return requested data to them. Sophisticated services may involve more than one server VM implying that the functionality of the operating system is distributed across several VMs.

4.1 Implementing a File Service

One of the more important services provided by an operating system is a file service which is implemented by a file system. The internal details of the file system can be hidden from the user by separating the logical structure of the file system from its physical structure. This is done by partitioning the file access facilities into two
functions, namely, the actual organisation of the file in terms of records on the disk as maintained by the file server and the file naming and protection mechanisms which are provided by the name server. The file server and name server are each implemented by a VM as illustrated in Figure 2.

The file server VM maintains a set of files, each file consisting of a linked list of disk blocks. A unique identifier (UID) is associated with every file. The file server implements operations or files such as read, open and write.

Clients refer to files by name. A name is a text string such as "/usr/joe/report5" which makes sense to a user. Such a file name is translated by the name server to a UID which is more convenient for the file system to handle. Directory structures are thus implemented by the name server. For reasons of efficiency there can be more than one name server in a distributed environment as long as a consistent database of names is maintained. The same resource can also be known to different users by different names if that is convenient.

4.2 User-level Processes

A user starts a session by logging in. This creates a single VM (with an associated screen and keyboard) which executes a single process—a Unix-like shell—capable of interpreting commands entered from the keyboard. Another session can be started by entering a command to create an additional VM. This may be repeated a prede-termined number of times. For instance a user can have a VM (session) for editing and another for compiling and testing a program. The keyboard and screen can be transferred from one VM to another to allow a user to switch between sessions.

A schematic view of the distributed system consisting of the kernel resident on each physical machine and an operating system supported by various VMs is given in Figure 2.
5 Performance of the Kernel

In order to determine the efficiency of the kernel a series of test programs were constructed. All tests were conducted on machines with a 25 MHz Intel-386 processor. Remote communication depends to a large extent on hardware speeds, but initial tests seem to indicate that a basic speed of 3 Mbits per second over an ethernet of 10 Mbits per second is attainable. Here we concentrated on measuring the efficiency of the kernel software: local interprocess communication overhead and the cost of the process and VM management facilities.

5.1 Interprocess Communication Facility

Two sets of test programs were used to evaluate the efficiency of the IPC facility. Firstly, messages of different lengths were transmitted from a process on one VM to a process on a different VM, both VMs residing on the same physical machine. The receiving process acknowledges receipt of the message by sending back an empty reply message. Such an interaction between two processes is called a message transaction[13] which is a simple form of a remote procedure call. Secondly, messages were transmitted between processes supported by the same VM.

An initial (perhaps naive) implementation of IPC yielded the performance figures shown in Table 1. The delay to transmit a single message is given in milliseconds and the communication bandwidth is given in terms of both Kbytes per second and a percentage of the maximum speed at which the machine can copy data from one memory area to another.

The reason why message passing between different VMs was slow in the initial implementation of the kernel is that messages were copied first from the sending VM to the kernel and then from the kernel to the receiving VM. Many context switches were necessary since long messages were copied one 4Kbyte physical memory page at a time. It was much simpler to transfer messages between processes on the same VM. In this case no context switches were necessary and the whole message could be copied in one go. Since the kernel is meant to support control software designed as several communicating VMs, it was mandatory to improve the efficiency of message passing between processes located on different VMs.

The hardware of the 386 allows a VM to have a 4Gbyte address space, each VM having its own page table. By reducing the maximum allowable size of the address space of a VM to 2Gbytes, only the lower half of the page table of each VM is used. The upper half of each page table can now be used to gain access to the source or destination address space of a message, whichever is appropriate. This technique allows us to use exactly the same code to handle message passing between processes on the same VM and processes on different VMs. It significantly improved the efficiency of message passing between processes on different VMs as shown in Table 2. Although the same code is used to transfer messages between processes on the same VM and processes on different VMs, the performance is different. If it is
<table>
<thead>
<tr>
<th>Message Size (Kbytes)</th>
<th>Delay (milliseconds)</th>
<th>Bandwidth (Kbytes per second)</th>
<th>Bandwidth expressed as a % of max. copy speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.69</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>3.36</td>
<td>298</td>
<td>2.0</td>
</tr>
<tr>
<td>2</td>
<td>4.00</td>
<td>499</td>
<td>3.3</td>
</tr>
<tr>
<td>4</td>
<td>5.27</td>
<td>759</td>
<td>5.0</td>
</tr>
<tr>
<td>8</td>
<td>7.93</td>
<td>1009</td>
<td>6.6</td>
</tr>
<tr>
<td>16</td>
<td>13.17</td>
<td>1215</td>
<td>8.0</td>
</tr>
<tr>
<td>32</td>
<td>23.65</td>
<td>1353</td>
<td>8.9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Message Size (Kbytes)</th>
<th>Delay (milliseconds)</th>
<th>Bandwidth (Kbytes per second)</th>
<th>Bandwidth expressed as a % of max. copy speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.51</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0.77</td>
<td>1299</td>
<td>8.5</td>
</tr>
<tr>
<td>2</td>
<td>1.03</td>
<td>1942</td>
<td>12.7</td>
</tr>
<tr>
<td>4</td>
<td>1.55</td>
<td>2572</td>
<td>16.8</td>
</tr>
<tr>
<td>8</td>
<td>2.60</td>
<td>3083</td>
<td>20.2</td>
</tr>
<tr>
<td>16</td>
<td>4.68</td>
<td>3419</td>
<td>22.4</td>
</tr>
<tr>
<td>32</td>
<td>8.84</td>
<td>3618</td>
<td>23.7</td>
</tr>
</tbody>
</table>

Table 1: Initial Implementation of IPC

kept in mind that page table entries are cached by the 386, this can be explained. The cache (translation lookaside buffer or TLB) is a buffer of fixed size and it is flushed on each context switch. When a message is transmitted between processes on the same VM a context switch is unnecessary and the TLB helps to speed up the task of copying the message. However, when messages are transmitted between processes on different VMs it is awkward to avoid a context switch and the extra amount of testing required to do so is not considered worthwhile. However, this possibility has not yet been explored thoroughly.

By carefully rewriting the procedure which is used to copy data in assembly language, an additional improvement in the transfer rate was possible—see Table 3. With messages of 32K in length a data transfer rate of 12400 KBytes per second can be sustained between two processes on the same VM. This is within 80% of the maximum speed at which the test machine can transfer data from one location to another in memory.

5.2 VM and Process Management Facilities

The creation of a new VM requires the establishment of a virtual address space, the creation of a process to execute code and the initialization of the VM. The minimum time required to create and terminate a VM is 12.1 milliseconds. This includes the time needed to: install an executable program on a VM, create and terminate a
<table>
<thead>
<tr>
<th>Message Size (Kbytes)</th>
<th>Delay (milliseconds)</th>
<th>Bandwidth (Kbytes per second)</th>
<th>Bandwidth expressed as a % of max. copy speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.80</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>1.24</td>
<td>810</td>
<td>5.3</td>
</tr>
<tr>
<td>2</td>
<td>1.66</td>
<td>1201</td>
<td>7.9</td>
</tr>
<tr>
<td>4</td>
<td>2.49</td>
<td>1606</td>
<td>10.5</td>
</tr>
<tr>
<td>8</td>
<td>4.16</td>
<td>1921</td>
<td>12.6</td>
</tr>
<tr>
<td>16</td>
<td>7.53</td>
<td>2125</td>
<td>13.9</td>
</tr>
<tr>
<td>32</td>
<td>14.24</td>
<td>2246</td>
<td>14.7</td>
</tr>
</tbody>
</table>

Table 2: Copying data directly

<table>
<thead>
<tr>
<th>Message Size (Kbytes)</th>
<th>Delay (milliseconds)</th>
<th>Bandwidth (Kbytes per second)</th>
<th>Bandwidth expressed as a % of max. copy speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.78</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0.89</td>
<td>1117</td>
<td>7.3</td>
</tr>
<tr>
<td>2</td>
<td>1.00</td>
<td>2000</td>
<td>13.1</td>
</tr>
<tr>
<td>4</td>
<td>1.22</td>
<td>3292</td>
<td>21.6</td>
</tr>
<tr>
<td>8</td>
<td>1.63</td>
<td>4893</td>
<td>32.1</td>
</tr>
<tr>
<td>16</td>
<td>2.47</td>
<td>6465</td>
<td>42.4</td>
</tr>
<tr>
<td>32</td>
<td>4.16</td>
<td>7692</td>
<td>50.4</td>
</tr>
</tbody>
</table>

Table 3: Hand-optimised implementation
<table>
<thead>
<tr>
<th>System</th>
<th>ratio of RPC to PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>182:1</td>
</tr>
<tr>
<td>Amoeba</td>
<td>200:1</td>
</tr>
<tr>
<td>HYBRID</td>
<td>295:1</td>
</tr>
<tr>
<td>Mach</td>
<td>416:1</td>
</tr>
</tbody>
</table>

Table 4: Efficiency of RPC compared to a simple procedure call

process and to allocate and deallocate a small address space. In general this figure depends on the size of the VM because the actual time required to create a new VM depends on the size of its address space and the size of the executable program.

The total time taken to create and terminate a process amounts to 0.55 milliseconds. This includes the allocation of a new process descriptor and stack for the process in the kernel, a run-time stack in user space, two context switches and deallocation of memory assigned to the process.

5.3 Comparing Simple and Remote Procedure Calls

The kernel has been in operation for some time and although some improvements are still possible, its performance is considered to be acceptable. Although we tried to make IPC operations as efficient as possible, remote procedure calls remain expensive when compared to simple procedure calls. The main reason for this seems to be the interaction with the scheduler. An interesting idea to be explored in this respect is upcalls[7]. Comparing the performance of different systems is normally meaningless unless the same hardware is used. Since this is seldom the case in practice, the best we could do was to compare the speed ratio between remote procedure calls and simple procedure calls. Similar measurements are available for a number of well-known kernels. As shown in Table 4 the HYBRID kernel is comparable in efficiency to the Amoeba, Mach and V kernels in this respect.

6 Conclusion

Several distributed systems were developed since the mid 1970's when RIG[2, 12], the first major distributed system, became operational. All these systems were based on small message based kernels. Chorus, Mach, Amoeba and the Stanford V system are representative of the current state of the art. Although they reflect different philosophies regarding communication and process management, all proved to be successful. The first goal of the HYBRID project was to exploit the best ideas from these experimental systems to produce a kernel which is useful as a practical tool to simplify the implementation of control software. Because most designers of control systems find it natural to work with machines of some kind, we
have designed a kernel which transforms the physical machine which is awkward to program into one or more virtual machines which are easier to program. VMs which can support multiple processes, are similar to team spaces in the V system, tasks in Mach and process clusters in Amoeba. VMs can be used to group cooperating processes together to optimise interprocess communication. The problem is to find the right abstractions which can hide irrelevant detail of the underlying physical machine without hiding its power. Whether VMs can do this will have to be seen, but design attempts undertaken so far are encouraging.

The functionality defined by the VM abstraction decouples a control system from the underlying hardware. It is therefore possible to use the same control software on different hardware platforms by porting the kernel.

Preliminary experiments led to a simple kernel for a Data General minicomputer by May 1989. We gained valuable experience regarding process and peripheral management during these early experiments. Relying on this experience we redesigned the kernel and a functionally complete HYBRID kernel for an Intel 386-based machine has been operational since the end of 1990. To date, the kernel has been used to develop a distributed operating system for research and educational purposes. Its performance is comparable to a commercial Unix system executing on the same hardware. The kernel has now been running without problems for about six months and appears to be reliable. Although currently used to support an operating system the kernel can be reused to support other applications. For example, a terminal concentrator can be designed as a number of cooperating VMs supported by a single physical machine.

7 Acknowledgements

The first device drivers for the kernel were written by Harry Lewis. As the first user of the kernel he detected a few subtle coding errors. The operating system which represented the first test of the kernel was developed jointly by William Howard and Harry Lewis.

References


An Implementation of Linda$^1$ Tuple Space under the Helios$^2$ Operating System.

Department of Computer Science, Rhodes University, Grahamstown, 6140 RSA
*Internet: cspc@alpha.ru.ac.za

Abstract
We discuss the implementation of Rhoda, our Linda-like Tuple Space server which runs under the Helios operating system. The approach analyses and partitions tuple space at compile time in order to reduce the run time overhead of tuple matching. The interaction between the concurrent processes and the tuple partitions is used as the basis for distributing the partitions and processes in the network. The paper presents some empirical results and discusses the suitability of the Helios nucleus for supporting the approach.

Keywords: distributed systems, parallel processing, transputer, Linda, Helios.

1. Introduction
The Linda programming paradigm is a simple and elegant approach to parallel processing, based on the concept of generative communication [GEL 85]. This is a form of communication in which an active message (or tuple) may be converted through process creation and evaluation into a passive value. Linda is not a language per se, it is a small set of control and coordination operations which can be imbedded into a programming language (typically one of the well known imperative programming languages) to introduce or enhance parallel capabilities.

At the center of the Linda programming model is a shared, associative memory called tuple space (TS). Objects called tuples are output to - and input from - TS by components of the application program. At the abstract programming level, TS is global to all components of a parallel program, even though they might be executing on individual processors which have no physical memory in common. Parallel components of an application program (processes or tasks) never communicate directly with each other, only with TS. Consequently, TS acts as a decoupling agent. This reduces program complexity by allowing parallel programs to be decoupled both spatially and temporally.

A tuple is a sequence of typed (actual or formal) fields, rather similar in concept to a parameter list. In addition to passive data values, the contents of a tuple may be a reference to active executing or executable code. Tuples are selected from TS by associative matching.

To communicate with tuple space, Linda provides a set of six primitive operations:

\[\textit{out}(t)\] output tuple \(t\) to the TS

---


$^2$Helios is a trademark of Perihelion Software Ltd., Somerset, England.
\textit{eval}(t) \hspace{1em} \text{evaluate tuple } t \text{ (This operation is similar to } \textit{out} \text{ in that it outputs tuple } t \text{ to } TS, \text{ but } t \text{ may be an active tuple whose result is yet to be evaluated.)}

\textit{in}(s) \hspace{1em} \text{input a tuple } t \text{ from } TS \text{ which matches the template } s \text{ (If no matching tuple is available, the requesting process is suspended until one becomes available. If more than one matching tuple exists in } TS, \text{ an arbitrary matching tuple is returned. The tuple is removed from } TS.)}

\textit{rd}(s) \hspace{1em} \text{read a tuple } t \text{ from } TS \text{ which matches the template } s \text{ (} rd \text{ is conceptually very similar to } \textit{in}. \text{ It returns a copy of a tuple without removing it from } TS.)

\textit{inp}(s) \hspace{1em} \text{and}

\textit{rdp}(s) \hspace{1em} \text{similar in function to } \textit{in} \text{ and } \textit{rd}, \text{ these operations are predicates which attempt to match a tuple } t \text{ to the template } s, \text{ and return a failure value immediately if no match is found. If the operation succeeds, both a tuple and a success value are returned.}

These operators communicate only with \textit{TS}, and none of the high-level system services which distributed operating systems usually superimpose upon their transport layers are provided. A Linda system may make use of the existing low-level transport layer provided by a distributed operating system, or may require a specialized transport layer to be written. In the former case, application programs should be unconcerned about the particular target architecture, and about whether they will run under an operating system or as standalone programs.

A number of informative articles on the use of the Linda approach to parallelism have already appeared in print, some of which are listed among this paper's references [CAR 89a] [CAR 89b] [GEL 88] [AHU 86]. We do not concern ourselves in this paper with presenting a suite of tutorial examples, or with persuading readers of the merits of this programming approach; we concentrate on implementation issues, assuming a rudimentary knowledge of the abstract programming environment presented by the Linda primitives, and a conviction of its value to parallel processing.

This paper represents a status report on an implementation effort underway at Rhodes University to build an efficient, distributed \textit{TS}-manager for transputer-based parallel processing systems in the Helios operating environment. To distinguish the experimental effort at Rhodes University from existing commercially available implementations of Linda, our system is known as Rhoda. For the purpose of this paper, the terms Linda and Rhoda are used interchangeably, although Rhoda is generally used to refer specifically to the Rhodes implementation.

2. \textbf{An overview of the Rhoda implementation}

A side effect of the high level of decoupling between parallel components of a Linda program is that efficiency becomes more of a concern of the implementation and less of a concern of the application programmer. This places pressure on the developers of a Linda implementation to provide an efficient transport layer which will allow \textit{TS} to be simultaneously visible to all components of the application program. A range of strategies can be used to implement a global \textit{TS} in a parallel processing environment in which processors do not have a shared physical memory. At one extreme, \textit{TS} could be stored at a dedicated central node which is accessed via a transparent message routing system. Even if run-time hashing is used to improve search
performance in this approach, delays caused by message routing can degrade the performance of the system, and a single centralized TS-manager can become a bottleneck which impedes massive parallelism. At the opposite end of the implementation spectrum, TS could be replicated in each processing node, and local TS-managers could transparently propagate changes through the network. A major encumbrance to this approach is the provision of a locking mechanism which ensures that program components wishing to remove tuples from TS are given exclusive delete access.

The Rhoda implementation under Helios uses a centralized TS model, but partitions TS with the view to reducing the run time matching overheads of Linda operations, and so that distributed TS-managers can be used to control a small (possibly localized) group of related tuples. A partitioned TS is in contrast to the Linda programming assumption of a single shared TS. This section provides a brief overview of the Rhoda compilation path, which adds additional housekeeping information to source programs to enable them to work with the partitioned model described in the remainder of the paper.

Figure 1 - Structure of the Rhoda compilation path.

Figure 1 depicts the compilation phases present in the Rhoda compiler. C is currently used as the host language for Rhoda. Apart from the normal C pre-processor, Rhoda makes use of a second pre-processor to compile and pass a list of all tuple operations, and the program components which issue them, to a tuple analysis module. This module analyzes TS interaction with components of the application program, to divide tuples into groups based on their structure, and to suggest an appropriate placement strategy for tuple groups and application program components in the processor network. The grouping of tuples is an integral feature of the Rhoda implementation, and is described in more detail below. By grouping tuples at compile time, a substantial matching overhead is avoided at run time. Distinct tuple groups also facilitate the distribution of TS in the distributed memory environment. The initial placement strategy of the Rhoda system divides a task force (application program components and TS-managers) into appropriate process clusters for placement on the processor network, in positions which will incur a relatively low inter-cluster communication cost. This aspect of TS analysis is described in more detail by de-Heer-Menlah [DHM 90].

The tuple groupings determined by the analysis module are used by the Rhoda pre-processor to translate ideal Linda syntax into concrete C syntax which opens, closes, and addresses file-like tuple partitions. A Rhoda program usually contains a number of components (for example, a master process and a worker process), which must all be present during tuple analysis. The output
of the pre-processor stage is a series of C programs, one for each unique parallel component of
the original source.

3. Partitioning tuple space

The syntax for tuple fields makes provision for actual fields in the form of constant values or run-
time expressions, and for formal fields denoted by program variables which are preceded by the
"?" character. The Linda input primitives provide tuple templates against which tuples placed in
TS by output primitives are compared. It is common practice for Linda programmers to use a
constant valued field to ensure a correct tuple matching. For example, the initial field of a tuple
is frequently a string literal. The matching process is potentially a computationally expensive
operation, and is an area in which efficient implementation is a crucial issue.

The tuple templates of Linda operations are matched by associatively searching tuples within TS
which have the same structure. Examples of syntactically correct, matching Linda primitive
operations might be:

\[
\text{out("element", 3, 4, value)} \quad \text{in("element", i + 1, j, ?result)}
\]

If the variables value, i, j, and result were all declared to be of the same type (integers for
example), then the two tuples manipulated by the above in and out operations would be regarded
as having the same structure, viz. a string constant followed by three integer fields. The actual
expressions (value, i + 1, and result) would contribute their current run-time values to the out
operation's tuple and the in operation's template. The formal field (result in the template used
for the in operation in this example) would return the value of a tuple whose first three fields
match those of the template. For example, if the values of i and j were 2 and 4 respectively, and
the tuple ("element", 3, 4, 12) were present in TS, then result would have the value 12 after
execution of the in operation.

It is possible to detect at compile time that a Linda input \text{in("row", ?i)} could be matched to any
of the following tuples

\[
("row", 4) \quad ("row", 10) \quad ("row", 500)
\]

with the consequent actual to formal assignment for the variable i.

It is likewise clear at compile time that the template ("result", ?i) will not match any of the
following tuples, no matter what the run time values of variables are, because the type, order, or
number of fields differ.

\[
("row", 6.42) \quad (j, "row") \quad ("matrix size", 50, 20)
\]

Nor will it match a tuple such as ("col", 4) whose type, order, and number of fields agree, because
the value of the compile time string literal field of the tuple and template differ.

Since operations on one tuple group can never match tuples in another group, the partitioning of
tuples into disjoint groups at compile time can be done safely. Tuple operations can first be coarsely classified into mutually exclusive groups based on their field structure (type, order, and number of fields). A subsequent finer partitioning can be done based on field information; tuples having the same field structure, but different compile time constant values in a particular field, cannot be matched.

Once compile time constants have been examined and tuple groups have been formed, the constant values are no longer of any use since all tuples (and tuple templates) within a particular group will have identical constant values in their common constant fields. Discarding such constant fields is a further compile time optimization. For example, Linda operations which refer to the tuples

\[ (\text{"row"}, i, j) \quad (\text{"row"}, i+1, j+1) \quad (\text{"row"}, ?m, ?n) \]

will be modified to calls to the same "row" tuple group using the tuples

\[ (i, j) \quad (i+1, j+1) \quad (?m, ?n) \]

Efficient searching and matching strategies can now be devised for particular tuple groups. Taken together, the dramatic reduction in the scope of a tuple search and the reduction in the number of (mostly string) fields provide a major improvement in the run time overhead of tuple matching. Analysis of the actual to formal relationships of the corresponding fields of a tuple template and its TS group can lead to further efficiencies in run-time matching. Zenith [ZEN 90] suggests a number of instances in which a general tuple matching algorithm can be reduced to a far simpler operation.

It is possible to take the TS analysis further by considering which components of the application program make use of each tuple group. This provides information for the placement of TS groups relative to the program components which they serve in the processor network, and allows an hierarchical TS dependency structure to be built, thereby facilitating an hierarchical naming scheme for the distributed TS. For example, a Linda application program comprising three parallel component processes, \( P_1, P_2, \) and \( P_3 \), coordinates its parallel activity using three different tuple structures which can be grouped at compile time into three independent tuple groups. All three processes make use of tuple groups 1, 2, and 3, as shown in Figure 2. The diagram illustrates how tuple groups can be connected to form a distributed network.

Figure 2 - Three parallel application processes referencing three tuple groups.
groups 1 and 2, while only \( P_2 \) and \( P_3 \) make use of tuple group 3. Figure 2 demonstrates the interaction of component processes and TS groups for this example, and figure 3 shows the hierarchical relationship which results from the partitioning of TS.

4. The Helios environment

Helios (PER 89) is a UNIX\(^3\)-like\(^4\), distributed, parallel operating system. The Helios nucleus, which must be present on all Helios processors, provides a small kernel (for managing message passing, hardware resources, and list handling) and a number of basic servers which integrate the processor into the global environment. Helios servers are based on the conventional client/server model, in which a server task manages a resource on behalf of its clients. The minimum set of servers required by a Helios processor includes a loader, a processor manager for managing the computing resources of the processor and for responding to requests to access executing tasks, and a number of I/O controller (IOC) processes. Additional operating system servers might be loaded on particular processors of the network to support specific facilities. These include the window server, the disk server, the RS232 server, the console server, the network server responsible for distributing and controlling the nucleus, and so on. Most importantly, Helios provides a server library facility which can be used to implement additional servers for the system using a standardized general server protocol.

To facilitate communication between distributed tasks, the process manager of the Helios nucleus spawns an IOC process for each new task, which acts as the task's intermediary with the rest of the system. The IOCs on one processor route requests to named objects on behalf of their tasks by referencing a central name table. If a name is present in the table, the IOC passes the request directly to the server whose port is represented in the entry, if not, a distributed search is initiated. Provided the name exists elsewhere, an entry is installed in the name table so that subsequent uses need not cause a search. Each physical link of the processor also has an IOC, responsible for handling distributed searches and requests from remote tasks to local servers.

Helios supports an hierarchical naming scheme for all objects in the network. Each sub-network (or cluster) is given a unique name, and the names of objects within sub-networks (processors, files, file systems, servers, tasks, and so on) must not conflict when they are identified by their

\(^{3}\)UNIX is a trademark of AT&T.

\(^{4}\)The Helios operating system includes a UNIX-compatible library, which is based on the POSIX standard [IEE 88].
position in the network hierarchy. All objects in Helios present a directory interface through which any information specific to the object may be examined and manipulated. This form of network addressing is a logical extension of the conventional hierarchical file system adopted by many operating systems. Most Helios commands which access the hierarchical directory structure are generic utilities which do not differentiate between different types of object in the hierarchy. Figure 4 is an example of the hierarchical naming scheme presented for a subnetwork. In this example, the cluster comprises three processors (00, 01, and 02) and an I/O server. The Helios nucleus on each processor comprises a tasks directory, a number of link IOCs, and so on. Objects within the tasks directory are the currently active tasks on that processor.

Network naming is a totally distributed service in Helios, and a distributed name server is at the heart of the naming scheme. It provides an hierarchy of names for an otherwise arbitrary topology structure.

Helios servers may be localized or distributed. All servers adhere to the same general server protocol (GSP). They are written as a set of calls to a distributed server library, plus a set of application specific functions [GAR 89]. The server library provides support for a message decoder and despatcher, which waits for messages on a specified port, validates them as GSP messages, and forks a worker process to execute a service procedure. The forking of a service procedure is an important aspect of Helios's support for distributed servers. The server essentially consists of the despatcher process until such time as a request arrives from the server's request port (looked up in the name table by the name server on behalf of a client process). To handle the request, the despatcher process spawns a separate process to execute the required function. This happens for each request. Normally, this process returns a reply at the end of the desired service and terminates. However, if the function performed by the spawned service process is an open operation (as in 'open a file'), the service process remains active after a reply has been sent, and acts as a proxy server for any stream messages which are directed to it, until it is closed.

5. The tuple space server under Helios

The hierarchical naming structure of Helios provides an ideal support environment for a TS which can be grouped in such a way as to expose hierarchical relationships. TS in the Helios-Rhoda
system has been implemented along similar lines to a directory based file server, in which each "file" corresponds to a tuple group capable of manipulating streams of tuples with the same type signature (as grouped by the tuple analysis module in the Rhoda pre-processor). By adopting the Helios environment, we gain directory and sub-directory structures, and their concomitant protection mechanisms, at no additional cost to the implementation; they are already part of the existing Helios server protocol and libraries. TS is implemented as a Helios server, using the standard GSP. The Rhoda TS server integrates very smoothly with rest of the Helios system because it honours this protocol, and the generic utilities which operate on other Helios objects are able to operate on TS structures as well. Each tuple group falls under the control of a TS-manager, but different tuple groups might be placed under the control of different TS-managers distributed across the network.

The Helios strategy of routing all GSP requests to a single port, and then spawning (by way of the despatcher) independent processes to service each of them, enables several clients to access the same server concurrently. In the TS server, such GSP protocols are occasional events, which open a tuple group and create a proxy process within the server to manage access to the tuple group on behalf of a particular client. Thereafter, Linda operations are reduced to direct communications between the client and the proxy process.

All client processes which produce or consume tuples with a particular type signature will open the same tuple group. To gain access to a tuple group, a client process must declare a tuple group descriptor, specifying a name for the group and a type signature for tuples which conform to the group. Thereafter, it is able to open the tuple group, use it, and close it again, simply by making appropriate TS server calls and supplying the appropriate name of the tuple group along with each such operation. The pre-processor prefixes each TS operation in the source code with a tuple group descriptor for this purpose. The first reference to a TS server initiates a dynamic network search and establishes a connection path between the client and its proxy service process, enabling the two to exchange messages without regard to the system topology. Thereafter the client process has a point-to-point virtual link to a dedicated proxy process, which manipulates the tuple group on its behalf, until it requests a close operation, at which time the proxy process terminates. Since several clients are able to access the same tuple group simultaneously, the TS proxy processes assume the responsibility for locking the tuple group and coordinating requests during operations which update the group.

Each tuple group within the TS server is a data structure which contains control information such as its name, size, number of clients, protection attributes, mode information, parent directory, a locking semaphore to ensure exclusive update access, and so on. It also keeps track of current tuple values, and keeps a queue of blocked clients together with their transaction templates.

To handle the blocking semantics of the Linda primitives rd and in, proxy server processes are suspended until a suitable tuple arrives. This has the effect of suspending the client as well, while it awaits a reply from the server. A TS proxy server handles an unmatched request by queueing it, along with a semaphore, in the waiting queue for the tuple group it supports. The proxy then suspends itself by waiting on the semaphore. Each time a new tuple arrives for a particular tuple group as a result of an output operation, the waiting queue for that group is searched, comparing the new tuple to pending requests. To satisfy the different semantics of the Linda in and rd
system has been implemented along similar lines to a directory based file server, in which each "file" corresponds to a tuple group capable of manipulating streams of tuples with the same type signature (as grouped by the tuple analysis module in the Rhoda pre-processor). By adopting the Helios environment, we gain directory and sub-directory structures, and their concomitant protection mechanisms, at no additional cost to the implementation; they are already part of the existing Helios server protocol and libraries. TS is implemented as a Helios server, using the standard GSP. The Rhoda TS server integrates very smoothly with rest of the Helios system because it honours this protocol, and the generic utilities which operate on other Helios objects are able to operate on TS structures as well. Each tuple group falls under the control of a TS-manager, but different tuple groups might be placed under the control of different TS-managers distributed across the network.

The Helios strategy of routing all GSP requests to a single port, and then spawning (by way of the despatcher) independent processes to service each of them, enables several clients to access the same server concurrently. In the TS server, such GSP protocols are occasional events, which open a tuple group and create a proxy process within the server to manage access to the tuple group on behalf of a particular client. Thereafter, Linda operations are reduced to direct communications between the client and the proxy process.

All client processes which produce or consume tuples with a particular type signature will open the same tuple group. To gain access to a tuple group, a client process must declare a tuple group descriptor, specifying a name for the group and a type signature for tuples which conform to the group. Thereafter, it is able to open the tuple group, use it, and close it again, simply by making appropriate TS server calls and supplying the appropriate name of the tuple group along with each such operation. The pre-processor prefixes each TS operation in the source code with a tuple group descriptor for this purpose. The first reference to a TS server initiates a dynamic network search and establishes a connection path between the client and its proxy service process, enabling the two to exchange messages without regard to the system topology. Thereafter the client process has a point-to-point virtual link to a dedicated proxy process, which manipulates the tuple group on its behalf, until it requests a close operation, at which time the proxy process terminates. Since several clients are able to access the same tuple group simultaneously, the TS proxy processes assume the responsibility for locking the tuple group and coordinating requests during operations which update the group.

Each tuple group within the TS server is a data structure which contains control information such as its name, size, number of clients, protection attributes, mode information, parent directory, a locking semaphore to ensure exclusive update access, and so on. It also keeps track of current tuple values, and keeps a queue of blocked clients together with their transaction templates.

To handle the blocking semantics of the Linda primitives _rd_ and _in_, proxy server processes are suspended until a suitable tuple arrives. This has the effect of suspending the client as well, while it awaits a reply from the server. A TS proxy server handles an unmatched request by queueing it, along with a semaphore, in the waiting queue for the tuple group it supports. The proxy then suspends itself by waiting on the semaphore. Each time a new tuple arrives for a particular tuple group as a result of an output operation, the waiting queue for that group is searched, comparing the new tuple to pending requests. To satisfy the different semantics of the Linda _in_ and _rd_
operations, a pass is made through the queue, locating each matching \textit{rd} transaction which can be completed, up to the first matching \textit{in} transaction. The \textit{in} transaction must also be completed, and will consume the new tuple. If there is no pending \textit{in} operation, the tuple is added to \textit{TS} in the normal way. Completion of a pending transaction is achieved by allowing the output primitive to complete its transaction, and then waking those proxy processes whose outstanding transactions can be satisfied.

Figure 5 illustrates the integration of the \textit{TS} server into the hierarchical Helios naming structure. In this example, a \textit{TS} server has been initiated on processor 00, and one or more client processes have opened two tuple groups, named \textit{rows} and \textit{results}. Client processes residing anywhere within the network are able to open either of these tuple groups, and a proxy service process will be spawned on processor 00 (within this \textit{TS} server) for each such request. So, process $A$, executing on processor 02, which uses both of these tuple groups, will cause two independent proxy processes to be spawned within this \textit{TS} server. Process $B$, executing on processor 00, which uses tuple group \textit{results}, will cause yet another proxy process to be spawned. There might well be additional \textit{TS} servers residing on other processors in the network and managing access to other tuple groups, provided their names do not conflict with the name of this server in the naming hierarchy. Processes $A$ and $B$ could well be making use of these additional servers as well.

The Rhoda system makes use of the Helios processor manager to implement the \textit{eval} primitive operation. The processor manager is a server which is present as part of the essential nucleus on all Helios processors. It sees to the creation and management of tasks on that processor, and is able to load and execute programs on behalf of clients executing on remote processors.

During the Rhoda pre-processing phase, each source function that is invoked by \textit{eval} is transformed into a free-stand alone executable program by encapsulating it in a suitable code skeleton. Since an \textit{in} or \textit{rd} template can never match an active tuple, tuples generated by \textit{eval} operations will be placed into their own active tuple groups. When a \textit{TS} server is invoked, it must be supplied with the names of the processors on which it may execute active tuples. The \textit{TS} server spawns a manager task for each such target processor, and establishes a link to that processor's processor manager. These manager tasks are responsible for monitoring the \textit{TS} server's active tuple group, and remotely invoking processes to evaluate tuples when necessary.
Remote program invocation is a relatively expensive operation, particularly if the executable code has to be fetched from a central filing system. The Rhoda implementation alleviates this overhead by modifying the skeleton that encapsulates eval-ed functions so that, once invoked, they repeatedly fetch and execute active tuples until a request for an active tuple matching their particular type fails. This has the same effect as reinvoking the function for every tuple of that type, but is considerably more efficient.

For monitoring purposes, each Rhoda TS server also provides statistical information, which appears to a client process wishing to monitor TS as just another set of tuples, which can always be read (i.e. they are created “on the fly” when they are requested).

6. Observations and conclusions

A number of desirable qualities are present in the Helios-Rhoda implementation:

The system is able to execute on any transputer network with an arbitrary topology.

With TS implemented as a distributed server (essentially present as part of the system nucleus on all processing nodes) no processors are dedicated to supporting TS, or are excluded by the presence of a TS-manager from supporting part of the application task force.

The division of TS into individually addressable tuple groups reduces the potentially expensive operation of associative matching to a far simpler operation, and provides a natural mechanism for partitioning TS space into distributable sub-spaces.

The hierarchical structure of TS partitions and the inheritance of the normal filing system security mechanisms allow concepts such as private tuple spaces and tuple spaces within tuple spaces [LEL 90] to be exploited.

Our approach differs from the Yale precompiler in that we view a parallel job as a single program comprising a number of sections. Our system requires that all the components are compiled and analysed together. Once the tuples have been partitioned and common fields have been factored out, the discarded information can no longer be retrieved unless the whole job is recompiled. By contrast, the Yale effort [CAR 90] supports separate compilation, and provides a pre-linking stage which analyses and specializes the tuple space access procedures. Their goal is to optimize the accesses, but to carry enough run-time information so that the original unoptimized data can be reconstructed. This will allow new participants to join the computation dynamically.

Some aspects of the system are still under development, notably the distributed TS-managers, and the system has only been tested with relatively small numbers of processing nodes (up to 16). The performance we have observed is encouraging. We have used the Rhoda implementation to support a number of parallel algorithms, including a state-space search and a 2-D FFT transformation. The Rhoda system is also being used as a platform for implementing a parallel version of a popular scientific and engineering matrix manipulation package [WEN 91], and as a means of parallelizing existing animation and graphics rendering applications. Figure 6 shows the almost linear improvement in speed obtained for an existing ray tracing application moved onto the Rhoda system, and for a queens placement algorithm, as the number of worker processes is

---

5This can be improved by program caching.
increased.

We have been happy with the performance of our 7S transport layer to date, and our experience also confirms the claims in the literature [CAR 89a] [CAR 89b] [GEL 88] [AHU 86] that it is easier to write parallel programs using the Linda model than it is with traditional tools. The high level abstract programming environment of the Linda operators has enabled us to think about parallelism in ways which were not always obvious when we were constrained by the concepts of semaphores and point-to-point messages.

Figure 6 - Performance of ray tracing and queen placement algorithms running on the Rhoda system, as the number of worker processes is increased.

From an implementor's point of view, Helios encourages a client/server programming model, and this has had a definite influence on our design. Without the presence of simple operating system mechanisms for creating and controlling tasks, topology independent message routing, and support for hierarchies of structures, we would not have envisaged the system as it is currently structured. This conviction is strengthened by our experience of designing a previous 7S prototype on a network of PC's running DOS, an environment which constrained our thinking severely. Moreover, Helios's Unix-like development environment and the ANSI-C language support have isolated us from the awkwardness of the transputer's underlying RISC-like architecture, and this has improved our productivity. It is unlikely that we would have made similar progress using TDS and Occam, the customary systems programming tools used with transputers.

References


Software Ltd.


The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System.

K. J. MacGregor
Dept of Computer Science, University of Cape Town,
Private Bag, Rondebosch 8000, South Africa
and
R. H. Campbell
Dept. of Computer Science, University of Illinois at Urbana Champaign, Urbana Illinois 61801, U.S.A.

Abstract
An object-orientated framework for the design of distributed virtual memory consistency protocols is presented. It is shown that custom designed protocols for different types of applications are easy to construct and use within this framework. Consistency protocols are shown to be useful in implementing atomic updates, and in controlling the assignment of pages to process.

An analysis of the performance of the Choices distributed virtual memory implementation is presented. It is shown that distributed virtual memory improves the performance of many applications by separating, and distributing the tasks of process management, synchronisation and paging. This performance improvement occurs despite the overhead in setting up and maintaining the coherence across the network. This is particularly noticeable when more processes than processors are active on a node. The results presented illustrate the effective improvements that can be achieved using correct coherence protocol and the current hardware restrictions limiting the performance of distributed virtual memory. In this it suggests further areas for research.

1 Introduction

Distributed virtual memory (DVM) exploits traditional virtual memory mechanisms to support a shared memory style of multiprocessing on a network of computers [Li86]. Shared memory parallel programs can be ported to a loosely coupled environment using DVM without change.

To maximise the number of parallel activities possible, local copies of the shared data are maintained on the machines involved. Some form of consistency among the various copies is guaranteed by means of consistency protocols. The protocols control the replication and invalidation of data within the system to support the required definition of consistency. Since they control page movement, they can be used to control the page assignment to network nodes.

In this paper, we study the Choices framework for the design of consistency protocols. This framework uses inheritance
in an object-orientated environment to permit new coherence protocols to be derived from existing ones. Three protocols are presented and the performance of two of them are analysed both with actual and synthetic timings. The timings illustrate the effectiveness of the various protocols for different applications, and in addition show how new protocols can be designed.

2. Overview of Choices.

Choices [Campbell87, Russo90] began as an investigation of the use of class hierarchies and object-orientated design for the construction of multiprocessor operating systems. All operating system concepts and components are implemented within the framework of a class hierarchy. Subclasses are used to encapsulate machine dependencies and to separate mechanisms from policy decisions. Choices is implemented in over 70,000 lines of C++ and includes code for virtual memory management, interrupt and exception handling, parallel processing, file systems, I/O, and networking. C++ supports class hierarchies and object orientated design without sacrificing efficiency.

Choices was designed on the Encore Multimax shared memory multiprocessor and is currently implemented on the Macintosh II, HP Precision, and Intel 386/486 computers.

2.1 Virtual Memory in Choices.

A Choices process [Russo88] executes in a Domain [Campbell88] which is a mapping of virtual address space to MemoryObjects. A MemoryObject is a logical collection of data. It is made accessible to the process by its MemoryObjectCache. A MemoryObjectCache maintains its own machine independent physical memory management information. A MemoryObject can be mapped into multiple Domains providing shared memory.

After a paged fault, the MemoryObject corresponding to the faulting virtual address is determined using the Domain. The RepairFault method is invoked on the MemoryObject to get the data at the virtual address. A local caching strategy, encapsulated entirely within the MemoryObjectCache is used to repair the fault. Subclasses of the MemoryObjectCache embody the ways in which a fault is repaired. This includes traditional paging, simple alteration of access (e.g. to implement copy on write), and getting the page from across the network (for DVM). A diagrammatic description of the sharing of memory in Choices is in Figure 1.

It is important to note that the Domain has no need to know or care how a MemoryObjectCache repairs a fault.
Figure 1: Memory sharing between Domains

Figure 2: A peer group of DMOC's
3. Distributed Virtual Memory

The DVM implementation consists of a number of parts: DVM setup methods, basic networking and the two most significant extensions to the virtual memory class hierarchy, the logical page information and the consistency mechanism. The two most important classes involved in the implementation of DVM, DistributedMemoryObjectCache and PageRecord are considered below.

3.1 Class DistributedMemoryObjectCache

An instance of DistributedMemoryObjectCache provides a local physical memory cache for the copy of shared data on a networked node. These DistributedMemoryObjectCaches form a peer group (Figure 2). Each DistributedMemoryObjectCache is responsible for locating and retrieving pages from its peers in order to repair virtual memory faults generated by processes on its node. Local copies of data can be paged to backing store on their respective nodes. This activity is also managed by the DistributedMemoryObjectCache.

3.2 Class PageRecord

The PageRecord class is the hub of the DVM implementation. Besides the traditional VM information inherited from the VM page record, page state and other information required for maintaining coherence is kept in instances of the class. The Choices consistency protocols are defined using state machines that are implemented in the PageRecord class. The methods of PageRecord and its subclasses correspond to the generation of and responses to events such as

- messages between nodes
- timeouts
- local read/write accesses
- process terminations

Existing protocols can be combined (using multiple inheritance) or subclassed to create new protocols. Thus new protocols can very easily be generated from the existing basic protocols, with most of the functions being inherited from the basic protocols and only the new features coded.

4. The Consistency Protocols

4.1 The Basic Choices Protocol

The basic Choices protocol [Johnston89, Li86] is designed to avoid the overhead associated with a heavyweight network protocol, therefore, it assumes a low-level, unreliable datagram service, i.e. delivery is not guaranteed. In addition it handles page to packet assembly/disassembly and recovery of lost packets. Consistency is maintained using a single writer/multiple readers discipline.
When writeable, a page resides on one machine. Read requests from other machines make the page read only, and copies of the page are sent to the requesters. A list of copy holders is maintained at the machine that originally had write access. This machine is designated as the owner of the page. A subsequent write request will be serviced by the owner, invalidating the readable copies and giving the writeable page to the requester. The requester then becomes the new owner of that page.

The message types in the basic protocol are:

- **GetWrite**: Sent to request the writeable copy (and ownership) of a page from the page's owner.
- **HereWrite**: Sent to transfer the writeable copy of a page to a node which has requested it.
- **AckHereWrite**: Sent to acknowledge a HereWrite.
- **GetRead**: Sent to request a read-only copy of a page from the page's owner.
- **HereRead**: Sent to transfer a read-only copy of a page to a node that has requested it.
- **GetUpgrade**: Sent to request write access (and ownership) of a page from the page's owner when the requesting node already has a read-only copy.
- **HereUpgrade**: Sent to transfer the ownership of a page to a node that has requested it.
- **AckHereUpgrade**: Sent to acknowledge a HereUpgrade.
- **Invalidate**: Sent to request that a node with a read-only copy of a page disallow all accesses to it.
- **AckInvalidate**: Sent to acknowledge an Invalidate.
- **OwnerHint**: Sent as a reply to a GetWrite or GetRead message when the recipient is not the owner of the page. The replying node provides "better" page knowledge to the requesting node.

Only the HereRead and HereWrite messages pass page data between the nodes. The other message types are control messages.

Reader and Writer processes can dynamically join or leave the group of processes using DVM.

This protocol is very general and does not provide any support to:

1. guarantee page assignment i.e.
   - retain pages until sufficient work is done with them.
   - order page usage among processes to enforce data dependencies.

2. guarantee atomic updates.

Without support within the protocol, the above can be achieved at the additional cost of unnecessary application knowledge, inefficiency and possible fatal susceptibility to untrustworthy processes.
4.2 Characteristics Determining Performance

The time overhead resulting from the protocol is a function of control and data packets generated and data structure processing. This can be represented as:

\[ O = N \cdot p_k + M \cdot p_g + F + V \cdot m \]

Where
- \( O \): protocol overhead.
- \( p_k \): Total number of packets transferred.
- \( p_g \): Total number of data packets.
- \( N \): Network transmission cost per packet.
- \( M \): Memory usage cost for data transfer per packet.
- \( F \): Fixed data structure processing costs independent of protocol events.
- \( V \): Data structure processing costs related to maintaining information on copy holders.
- \( m \): Number of machines.

Not all of this overhead reflects directly on the total time taken by the application, since some events are handled concurrently with the application. Performance also depends on page assignment. The basic protocol determines the overhead of events.

For the basic protocol events Read, Write, UpgradeO, and UpgradeNO (change from read copy to write when the node is the page owner and not the page owner respectively) the maximum overhead, assuming a 4k page size and using Ethernet as a transport mechanism with a 1.5k packet size, is:

**Read**: GetRead request (1 packet) + HereRead (3 packets)
\[ = 4N + 3M + V \]

**Write**: GetWrite (1 packet) + Invalidates ((m - 1) packets\(^1\)) + AckInvalidates ((m - 1) packets) + Domain flush (Fixed cost) + HereWrite (3 packets) + AckHereWrite (1 packet)
\[ = 5N + 3M + F + (m - 2)(V + 2N) \]

**UpgradeO**: Invalidates ((m - 1) packets)
\[ = (m - 1)(V + 2N) \]

**UpgradeNO**: GetUpgrade (1 packet) + Invalidates ((m - 1) packets) + AckInvalidates ((m - 1) packets) + Domain flush (Fixed cost) + HereUpgrade (1 packet) + AckHereUpgrade (1 packet)
\[ = 3N + F + (m - 2)(V + 2N) \]

4.3 A Locking Protocol

This protocol guarantees atomic update by denying requests for a locked page. If the basic protocol and test-and-set locks are used, it is not possible to guarantee that a page will remain; circumvention of the convention by untrustworthy processes is possible.

A response message **Retry** is returned to any requesters. Processes receiving this message can either sleep or send the request again. This defines the lock to be a sleep or

\(^1\) 0 if \( m \leq 2 \)
spin lock respectively. For a sleep lock, a queue of requesters is kept to send a wake-up. Sleep locks minimise network loading but centralise wake-up information and blocking primitives. A spin lock provides non-blocking synchronisation. Response to Retry can be handled differently on each requester. Network loading is minimal if the retry interval is made sufficiently large so that the page is available in the next request. This protocol can also be used as an ordinary lock to guarantee order of page use to satisfy data dependencies. Such usage, however, is restricted to processes executing in different Domains. It is also possible to lock pages for a given amount of time, to satisfy the page requirements of an application when they can be predicted beforehand or determined by some heuristic.

4.4 A Delay protocol

The Delay protocol is a method for retaining pages for a sufficient time without specific knowledge. In the basic protocol a request is serviced immediately. Two, or more, processes writing simultaneously could cause a page to thrash from machine to machine, without any useful work being done.

A threshold time, after which external requests for copies of the page are serviced, can improve the ratio of useful time to page transport time in all except pathological cases. Requests arriving during this time are queued and serviced later. The threshold time value is decided by the current activity within the page and the number of requests for that page in order to generate a page assignment that reduces the page movement and increases page usage. Threshold time also reduces the possibility of live-lock, where a page is shuttled between some machines, and others never get access. Whereas the locking protocol is user dependant enforced by lock and unlock system calls the delay protocol is dependant on the transmission speed of the network, the processor speed of the various computers and the level of access. The range of applications that can use this protocol vary from numerical to artificial intelligence which show different characteristics in their pattern of page use, the numerical being the more predictable and as such the easier to optimise. This protocol is still being studied to determine a suitable heuristic to decide the threshold time, particularly for commercial systems usage. As its characteristics are not significantly different from the locking protocol it is not included in the timing analysis.
4.5 Variations

The protocols can be altered to trade off characteristics such as resiliency to packet loss, network loading, etc. For example, in the basic protocol, only an approximate knowledge of the owner is kept on all nodes except those that have read/write access. Thus, requests from a node that does not have a copy of the page can end up at a node that is not the actual owner. One of two actions can be taken; forward the request to the current value of the owner, which is approximate, until the actual owner is reached [Forin89, Li86], or indicate the possible owner to the requester for an explicit send by the requester. In the former strategy, the number of messages is less than in the latter. However, the latter is more resilient to network packet loss.

5. Empirical Results

Two algorithms, matrix multiplication and Producer Consumer, were chosen as they exhibit extremes of page conflicts. The applications were run on 2 Encore Multimaxes each with 4 processors connected by Ethernet.

5.1 Matrix Multiplication

A simple 3 loop matrix multiplication program was used. In the case of the Lock protocol each page was locked when calculating the row on that page. Table 1 shows the timings for the two protocols. The results are summarised below:
- Matrix multiplication time for DVM is comparable to that for shared memory.
- DVM offers access to a greater number of processors, and gives better performance when the number of processes on a single machine exceeds the number of processors.
- Paging behaviour is drastically altered by access patterns.
- The locking algorithm performs slightly better by saving page transfers.

5.2 Producer Consumer

The producer locks the buffer and fills it, then releases the lock. The consumer then acquires the lock and empties the buffer. The processing time is enough to force Retry. Table 2 illustrates the timings for Producer Consumer. The Lock protocol is significantly more efficient than the basic protocol even when spin locks are used – 33.4% of the basic protocol time for a buffer up to 4k in size and 55.4% of the basic protocol time for a buffer of 10k. When the buffer size increases to over 60k the basic protocol is more efficient. The amount of network traffic generated by the Lock protocol is also less. For larger packet sizes, and
longer fill/empty processing times, traffic could be further reduced by using a sleep lock.

### Table 1: Comparative timings in sec, for matrix multiplication, estimated accuracy \( \pm 1 \) sec

<table>
<thead>
<tr>
<th>kij</th>
<th>ijk</th>
</tr>
</thead>
<tbody>
<tr>
<td># Processes</td>
<td>Shared</td>
</tr>
<tr>
<td></td>
<td>Basic</td>
</tr>
<tr>
<td>2</td>
<td>89.2</td>
</tr>
<tr>
<td>4</td>
<td>44.6</td>
</tr>
<tr>
<td>8</td>
<td>45.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ijk</th>
</tr>
</thead>
<tbody>
<tr>
<td>186.7</td>
</tr>
<tr>
<td>96.5</td>
</tr>
<tr>
<td>51.9</td>
</tr>
</tbody>
</table>
The Intel 80386/80486 can have variable page sizes and a 1k page for memory swaps could prove more efficient with an Ethernet transfer medium. This latter case however changes the access characteristics of the applications as there would be fewer page conflicts as the amount of data shared between nodes would be reduced. A 1k page is incompatible with most Unix type file systems. The smaller page size requires investigation if DVM is to be used as a cheap implementation of parallel processing. DMA accesses from the user data area onto the network using scatter-gather hardware would also reduce the number of memory copies. Increases in processor speed, and network transmission speed would only provide marginal improvements.

Table 3: Basic Operation timings, in ms for varying system component speeds (for 2 machines) estimated accuracy ± 1ms

<table>
<thead>
<tr>
<th>Event</th>
<th>Measured</th>
<th>Memory</th>
<th>Network</th>
<th>Processor</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>39.9</td>
<td>27.8</td>
<td>30.4</td>
<td>33.7</td>
<td>8.2</td>
</tr>
<tr>
<td>Write</td>
<td>48.2</td>
<td>36.2</td>
<td>36.4</td>
<td>41.2</td>
<td>1.2</td>
</tr>
<tr>
<td>Upgrade0</td>
<td>19.4</td>
<td>19.4</td>
<td>12.3</td>
<td>18.4</td>
<td>1.2</td>
</tr>
<tr>
<td>UpgradeNO</td>
<td>11.1</td>
<td>11.1</td>
<td>6.3</td>
<td>11.0</td>
<td>6.2</td>
</tr>
</tbody>
</table>

7 Conclusion

Performance of applications operating in a DVM environment is sensitive to their locality properties. To achieve the best performance it is necessary to choose an appropriate consistency protocol. This paper demonstrates a versatile framework for the design of such protocols. It is easy to specialise existing protocols or combine them to beat a custom designed protocol. The remainder of the VM system supports the creation of independent policy decisions for the different shared areas. It has also been shown that consistency protocols can be used to achieve more than consistency. The Lock protocol provides a way of guaranteeing atomic update, by preventing the movement of a page. These page locks can also be used as distributed locks, however their use is restricted to between Domains. When used as distributed locks they are cheaper than test-and-set locks. This protocol can also be used to assign pages, when predictable by the application. The Delay protocol attempts to optimally assign pages based on page usage, retaining pages on a machine. This is achieved without any knowledge of the application. The protocol also reduces the possibility of livelock problems. Other protocol variations to trade off the characteristics that affect performance are also indicated.
The experimental results are limited to the Encore Multimax however they show the hardware restrictions currently limiting the performance of DVM. The results show adequate promise that further investigation on the effects of page size, transfer medium and application performance should be carried out to achieve the optimal performance of applications in a DVM environment.

References


Concurrency Control Mechanisms for Multidatabase Systems

Andrew Deacon

Department of Computer Science
Stellenbosch University

Abstract

In conventional systems, transaction management has been thoroughly investigated and is now considered to be well understood. Transaction management in multidatabase systems, however, is less well understood. A multidatabase system (MDBS) is a type of distributed database system built from a collection of different centralized database systems (DBSs) located on sites in a computer network. The MDBS design goals differ from those of conventional DBSs in that a compromise is sought between the seemingly contradictory goals of creating a logically integrated database and allowing each component DBS to continue to function autonomously from the MDBS.

The conventional concurrency control correctness criterion for a DBS is serializability. It is, however, not always possible to guarantee serializability in a MDBS using the conventional approach. Several seemingly different solutions have been proposed. Using the nested transaction paradigm we provide a simple model which is used to develop a number of new MDBS schedulers by applying existing theory and concepts. The utility of our model is further illustrated by using it as a framework for explaining and justifying existing solutions, including those with weaker correctness criteria.

Key words and phrases: concurrency control, multidatabase systems, multi-level nested transactions, top layer schedulers, transaction processing model

1 Introduction

Database systems (DBSs) have effectively replaced file systems as the means of storing important and complex information used by an organization [Sib76]. The DBS simplifies the management of information by enabling data to be shared between users and providing control over different data copies. An important concept in DBSs is that of a transaction. A transaction is an execution of a program that accesses a shared database. The only way users are able to interact with the DBS is by creating and then submitting transactions. Transactions must appear to execute atomically. Ensuring the atomicity of transaction executions are the concurrency control and recovery problems. These problems have been extensively studied using a simple DBS model with considerable success (see [BHGS76] for example).

With the proliferation of DBSs has come the need, particularly for large organizations, to access multiple databases. The drive for vendor flexibility and the suitability of different environments for specific application requirements results in these databases being stored on multiple heterogeneous and autonomous systems. A multidatabase system (MDBS) is a type of distributed DBS that is built up from a collection of heterogeneous centralized DBSs located on sites in a computer network. One of the MDBS design goals is to create a system that hides aspects of where the data are stored and how data are retrieved from or updated in any DBS. Thus the illusion is created of a logically integrated DBS. Each component DBS must be able to be inte-
grated into the MDBS without modifications and still be able to function autonomously.

Since an MDBS also supports the DBS transaction concept, similar types of concurrency control correctness criteria should ideally be adopted. Here we will discuss some of the problems in adopting these criteria and their implications. The presentation is not technical, our goal being to introduce a new transaction processing model for MDBSs and to show the utility of viewing an MDBS as a multi-level nested system.

First the traditional DBS transaction processing model is summarized. Next we highlight some problems when the simple DBS model is applied to MDBSs. The multi-levelled nested model is developed to provide solutions to the problems. Unfortunately implementations are difficult and expensive and in some applications the traditional correctness criterion may be too restrictive. In Section 4 some of the proposed weaker correctness criteria are discussed. Lastly problems relating to the integrating of component DBSs are discussed.

2 Concurrency Control in Conventional Database Systems

The correctness of any system must be defined relative to users' expectations. Intuitively, a system is correct if it does what users want it to do. It is generally accepted that the correctness specification of concurrent executions in the context of DBSs is serializability [BHG87]. From a user's perspective, serializability essentially ensures that a sequence of atomic operations can be performed such that the overall effect is the same as would result if each user transaction was executed, in some order, one after the other [Pap79]. This notion of correctness is appealing, as the DBS need not know anything about the computations to be performed by a transaction.

First some important concepts in serializability theory are covered. Next we discuss the different scheduling mechanisms used in DBSs. Some variations, enhancements and extensions to the basic scheduling mechanisms are then discussed.

2.1 Serializability Theory

From the viewpoint of serializability theory, the only operations of a transaction that need to be modeled are those that access the database, namely the Read and Write operations. A database is viewed as a collection of named data items. Each Read and Write specifies the name of the data item read and written, respectively. In addition each completed transaction contains either a Commit or Abort as its last operation, indicating whether or not the transaction terminated successfully. We use $r_i[z]$ to denote a Read operation, issued by transaction $T_i$, on data item $z$. Similarly $w_i[y]$ denotes a Write operation, $c_i$ the Commit of transaction $T_i$, and $a_i$ an Abort.

A history indicates the order in which operations, submitted on behalf of a transaction, were executed relative to one another. As some operations may be executed concurrently, a history is defined in terms of a partial order, as is a transaction. If two operations operate on the same data item and one of the operations is a Write, then they are said to conflict. A history must specify the order of all conflicting operations that appear in it. Two histories are equivalent if firstly they are defined over the same set of transactions and have the same operations, and secondly they order conflicting operations of committed transactions in the same way [BHG87].

A complete history $H$ is serial if, for every two transactions $T_i$ and $T_j$ that appear in the history, either all operations of $T_i$ appear before all operations of $T_j$ or vice versa. A history $H$ is serializable (SR) if, considering only committed transactions, it is equivalent to a serial history $H_S$. Since serial executions are correct and because every SR execution, by definition, is equivalent to a serial one, every SR execution is also correct. The concurrency control problem is thus ensuring that all executions are SR [BG81].

Determining whether or not a history $H$ over $T = \{T_1, \ldots, T_n\}$ is SR is achieved by analyzing a graph derived from the history. This graph is called a serialization graph (SG). The SG for $H$ is a directed graph in which nodes represent the committed transactions in $H$. The edges are all $T_i \rightarrow T_j$ ($i \neq j$) such that one of the operations
of \( T_i \) precedes and conflicts with one of the operations of \( T_j \). The **Serializability Theorem**, given in terms of these definitions, states that a history \( H \) is SR if and only if the SG of \( H \) is acyclic [BHGS].

### 2.2 Concurrency Control Mechanisms

The question may be asked why transactions are not executed serially. Current computer systems have two basic types of storage: volatile and stable. Volatile storage is fast but expensive and data will be lost in the event of a power failure. Stable storage is cheap and usually has a large capacity but is very slow. While one transaction is accessing stable storage there will be an opportunity for the system to perform other operations. Greater multiprocessing is required to improve performance.

Concurrency control mechanisms must guarantee that all executions are SR while still being efficient. This is not achieved by adding synchronization mechanisms like semaphores, but by designing an algorithm, called a scheduler, that monitors the execution and intervenes to change the execution order of operations whenever necessary. The scheduler must perform its task at speeds comparable to the execution of the program [Pap]. What is submitted to the scheduler is a sequence of accesses to the shared data.

#### 2.2.1 Two-Phase Locking

Locking is the most widely used concurrency control mechanism. This is because of its simplicity and generality. Each data item has a lock associated with it. Before a transaction \( T_i \) may access a data item, the scheduler first examines the associated lock. If no transaction holds a conflicting lock, then the scheduler obtains the lock on behalf of \( T_i \). If another transaction \( T_j \) holds a conflicting lock, then \( T_i \) has to wait until \( T_j \) gives up the lock. That is, the scheduler will not give \( T_i \) the lock until \( T_j \) releases it. The scheduler thereby ensures that only one transaction can hold a conflicting lock at a time, so only one transaction can alter the data item at a time.

To guarantee serializability all locks must be acquired before any are released [EGLT]. Thus there are two phases, first acquiring locks and then releasing them. The locking mechanism is thus called **Two-Phase Locking** (2PL). If locks are all released when the transaction commits then the execution is said to be **strict** [BHGS]. Strict executions simplify recovery by preventing cascading aborts.

An unfortunate property of 2PL schedulers is that they are subject to deadlocks. A **deadlock** is a circular waiting situation, involving two or more transactions, such that each transaction is waiting for another to complete. Deadlock can be described precisely in terms of a wait-for graph. A **wait-for graph** is a directed graph with transaction identifiers forming the nodes. An edge from transaction \( T_i \) to \( T_j \), denoted \( T_i \rightarrow T_j \), represents the fact that \( T_i \) waits for \( T_j \). The existence of a cycle in the wait-for graph indicates a deadlock situation. Maintaining and checking a wait-for graph is expensive; thus more efficient deadlock resolution mechanisms are used in practice [BHGS].

#### 2.2.2 Timestamp Ordering

In timestamp ordering (TSO), the DBS assigns a unique timestamp to each transaction. Before another timestamp can be assigned, the counter is changed, thus ensuring that all timestamps are unique in the system. The DBS attaches a transaction’s timestamp to each operation issued by the transaction. The TSO scheduler orders conflicting operations according to their timestamps. By enforcing this rule, we are ensuring that every pair of conflicting operations is executed in timestamp order. Thus, a TSO execution has the same effect as a serial execution in which the transactions appear in timestamp order. With TSO deadlock is avoided since no two transactions can be waiting for one another if the TSO rule is obeyed.

#### 2.2.3 Serialization Graph Testing

**Serialization graph testing** (SGT) schedulers maintain the SG of the history that represents
the execution it controls. The scheduler attains SR executions by ensuring the SG it maintains always remains acyclic. The SG maintained by the scheduler differs from the one described in Section 2.1 because many older transactions that cannot be part of cycles are not represented and active transactions are included to prevent cycles being formed.

2.2.4 Variations on the Basic Schedulers

Variations on the basic schedulers discussed above have been developed. This permits a scheduler to balance the costs between delaying executions and having to abort transactions whose executions are not SR.

An optimistic scheduler is based on the premise that it is sometimes easier to "apologize" than to "ask permission" [Her90]. In optimistic approaches, a transaction is allowed to execute without any operations being delayed, and then validated before completing to determine whether a conflict has occurred. In the case of an inconsistency, a transaction is aborted and then resubmitted. Any optimistic scheme is cost effective only if validation succeeds sufficiently often.

In a pessimistic scheduler, operations are synchronized before they are executed, to produce a SR execution. Pessimistic schedulers operate under the assumption that transaction conflicts will be frequent and if an inconsistency can occur, it will occur. These mechanisms avoid the overhead of aborting and redoing transactions which, if committed, would have resulted in a non-SR execution.

Pessimistic schedules can be further classified as aggressive or conservative. An aggressive scheduler tends to avoid delaying operations and will try to schedule operations as soon as possible. The opportunity to reorder operations may be lost, resulting in transactions having to abort. A conservative scheduler tends to delay operations, giving the opportunity to reorder operations received out of SR order, thus reducing the likelihood of causing a transaction to be aborted.

2.3 Multi-level Concurrency Control Mechanisms

A useful approach to design and description of complex DBSs is to decompose the system into a hierarchically organized collection of levels [BSW88]. This has been widely used in a number of systems where record level locking is crucial to obtain satisfactory throughput rates [ABC*76, PSS*87]. While page operations are the most efficient means of accessing stable storage, using page locks would result in an intolerable number of conflicts. By application of the multi-level paradigm, systems can be designed that process multiple non-conflicting record operations in parallel [BBG89].

In order to resolve page level conflicts, additional short term locks are needed. Since these locks are used only for isolating single record operations, they can be released after each record level action without violating serializability. Unfortunately recovery is made more complex since page-before images cannot be used to undo a partial execution. Consequently recovery actions must be performed by inverse operations on a level higher than the operation being undone [BSW88, PSS*87]. Cascading aborts are thus inevitable.

2.4 Distributed Schedulers

In a single processor system, the reliability of the system is closely related to the reliability of its processor. In a distributed system that consists of sites with their own processors, it might be possible to execute a transaction even after sites have failed by using the resources of other sites. 2PL can easily be used to build a distributed scheduler. In the remainder of this article we will assume Strict 2PL is used unless stated otherwise. Each site has a scheduler, based on 2PL, that manages data items stored locally. For reasons of simplicity we will assume that there is no data replication.

The schedulers at all sites, taken together, constitute a distributed scheduler. The task of the distributed scheduler is to process the operations submitted in a (globally) SR and recoverable
manner. In 2PL, a Read(z) or Write(z) is processed when the appropriate lock on z has been obtained, which only depends on what other locks on z are presently owned. Thus the scheduler at each site has all the information needed to decide when to process an operation, without requiring communication with other sites. However the release of locks is not as simple. To enforce the two-phase rule, a scheduler cannot release the locks of a transaction at one site until it knows that no other locks will be obtained at another site. If Strict 2PL is used, communication between sites can be avoided by integrating the locking release in the Commit.

In distributed DBSs, the Commit and Abort operations differ from those in centralized systems, in that these operations have to be collectively executed by all sites involved in an execution and still appear atomic to the user. In the absence of a general mechanism to manage replicated data, the only non-trivial problem is that of consistent termination [BH87]. An algorithm that ensures consistent termination is called an atomic commit protocol (ACP). Guaranteeing the atomicity of a distributed transaction is to ensure that the entire transaction is either unanimously aborted or unanimously committed. A mixed decision can result in an inconsistent database [Ske81].

The most widely used ACP is the two-phase commit protocol. If all operations of T do executed successfully at a site, a precommitted state is entered, and a message is sent to the transaction’s commit protocol coordinator. This forms the first phase of the protocol. Locks are then held until either a Commit or Abort operation is received. Only when the coordinator has determined that all sites where operations were executed are in a precommit state, is the Commit operation sent to each site. Otherwise Aborts are sent to all sites in a precommit state. This forms the second phase of the protocol. If, as a result of a failure a decision cannot be made to either commit or abort, locks must be held until a decision can be made.

3 Concurrency Control in Multidatabase Systems

Implementing an MDBS that executes transactions serially is straightforward. For example, all transactions may be executed one after the other or updates may be executed off-line. For many applications this is unsatisfactory. The solution is to permit concurrent executions. However guaranteeing that concurrent transaction executions are SR is difficult in an MDBS because MDBSs are not aware of all transactions executed on the shared data and implementing an ACP may not be possible [GL84]. Because of these problems the majority of MDBSs have limited support to distributed read-only transactions or updates at a single component DB.

In the following subsections we investigate problems in developing a correct MDBS scheduler. First we consider implementing a distributed 2PL scheduler in an MDBS where component DBSs enable an ACP to be implemented. Concurrency control mechanisms other than locking could equally well have been used, but since locking is the most widely used mechanism in DBSs, we limit our discussion to a locking mechanism. Next a broader class of schedulers is developed where the component DBSs that do not support an ACP can be incorporated into the MDBS with the limitation that they be used only for retrieving data or that only one component DBS is updated.

3.1 Distributed Two-Phase Locking

Implementing the distributed 2PL mechanism discussed in Section 2.4 is only possible if the component DBSs have mechanisms that permit the construction of an ACP. On receiving a global transaction, the MDBS decomposes it into a number of subtransactions that can each be executed at a single component DBS. The results of these subtransactions are then integrated by the MDBS, by executing join, union, select and project operations, if we assume the relational model is used. Intermediate results may be sent to other component DBSs where additional subtransactions have to be executed. A final result
for the transaction is created and returned to the user. All locks acquired during the execution must be held until commitment. Typically each component DBS will support a Pcommit command that forms the first phase of the ACP.

As with all locking based mechanisms, deadlock is possible. However, the conventional solutions to the deadlock problem cannot be used in an MDBS as the following example illustrates.

**Example 1**
Consider an MDBS with two component DBSs A and B. Data items x and y are at component DBS A and z and u are at component DBS B.

Consider two global transactions: G1 which updates x and y; while G2 updates y and z. In addition let two local transactions execute: L3, submitted to component DBS A, reads x and y; and L4, submitted to component DBS B, reads z and u.

We assume that the two global transactions are each decomposed into two subtransactions which will be denoted G1A, G1B, G2A, and G2B, where the first element of the subscript denotes the transaction identifier and the second the component DBS where it is executed. To ensure that the execution is SR and recoverable, each subtransaction issues a Pcommit after executing the last Write operation.

Consider the following execution where at A, G1A executes and is in a precommit state. Then L3 obtains a read lock on y and waits for G1A to release its write lock on x. Now G2A is also prevented from executing since it must wait for L3 to complete before it can obtain the write lock on y. At B the reverse situation occurs, G2B is in a precommit state and L4 holds a read lock for u which blocks G1B. Thus neither G1 nor G2 is able to execute to completion which causes deadlock.

In a conventional system it would have been possible to construct the following wait-for graph to detect the deadlock:

```
  L4
   \   \ 
   G1  G2
   /   / \
  L3
```

This deadlock situation cannot be detected by the MDBS by constructing a global wait-for graph because the MDBS is not aware of the local transactions.

One solution is to modify the component DBS to enable the MDBS to detect global deadlocks [DL87]. This approach is undesirable because it requires modifications to the component DBS, which is generally not possible.

Various algorithms have been proposed that prevent global transactions from entering states in which global deadlock may occur. These approaches have relied on maintaining an acyclic graph representing the component DBSs at which subtransactions have been executed. Before another subtransaction can be issued a check is performed to ensure that the new graph will not become cyclic [RS82, BS88].

Timeouts provide a very simple deadlock resolution mechanism that has been widely implemented, particularly in commercial systems [BHG87].Timeouts will cause at least one transaction to eventually abort. An advantage of using a timeout mechanism over a protocol that prevents deadlock is that it does not rely on the correct functioning of all components [RS82]. However, if the timeout period is not carefully chosen and is too short, unnecessary transaction aborts are possible. Unnecessary delays may result if the period is too long. A knowledgeable person is typically required to select the timeout period.

### 3.2 Top Layer Schedulers

The distributed 2PL scheduler discussed above cannot be used in a network where data are retrieved from component DBSs that do not support a Pcommit command, because otherwise the execution may not be equivalent to a two-phase execution. We consider how non-SR executions may be produced involving distributed read-only transactions. If these inconsistent reads are used to update the component databases then the MDBS may cause the databases to become inconsistent.
3.2.1 Inconsistent Reads

Although all transaction executions at the component DBSs are SR, this does not guarantee that all global transactions are SR as the next example illustrates [BS88].

Example 2

Consider the same database described in Example 1. Assume that two global transactions: \( G_1 \) which reads \( x \) and \( u \); and \( G_2 \) which reads \( y \) and \( z \). There are no conflicting operations between the global transactions \( G_1 \) and \( G_2 \) according to the definition in Section 2.1.

In addition consider two local transactions: \( L_3 \), submitted to component DBS \( A \), reads and writes \( x \) and \( y \); and \( L_4 \), submitted to component DBS \( B \), reads and writes \( z \) and \( u \).

The histories in Figure 1 represent possible executions of the local transactions together with subtransactions, executed on behalf of global transactions, at the two component DBSs. In addition, the SG for the execution at each component DBS is given. The notation \( x_A \) is used to indicate that data item \( x \) is stored at \( A \).

If both local and global transactions were executed together in a conventional DBS, the following SG would have been created:

\[
SG(H) = G_1 \quad \quad \quad \quad G_2
\]

\[
L_3 \quad \quad \quad \quad L_4
\]

The presence of the cycle implies that the above execution is not SR. The two global transactions each reflect results that could be obtained if transactions were executed serially, but it would be impossible for the results to have been produced together. \( \Box \)

Thus any two subtransactions executing together at the same component DBS can conflict, regardless of which data items were read. This was not unexpected since the executions of the global transactions are not two-phase. The 2PL policy requires that no locks may be released until all locks have been acquired. In Example 2, locks were released by each global transaction at one component DBS before each acquired additional locks at a second component DBS.

3.2.2 Multi-Level Model

The history, defined in Section 2.1, models transaction executions in conventional DBSs very well. However, it does not model MDBS transaction executions as successfully. The fact that each global transaction is executed as one or more subtransactions is not captured and thus cannot be exploited by a MDBS scheduler. To capture this information we define MDBS histories in a similar way to the nested history used in multi-level nested systems [BSW88, BBG89]. This view of MDBS transactions as being nested has been recognized by a number of authors, including Ries & Smith [RS82] and Gligor & Luckenbaugh [GL84]; however, they did not use this model to develop any new schedulers.

MDBS histories are represented by separate histories for each level. The global transactions represent top level transactions, with the second level being the local transactions and subtransactions submitted to the component DBSs. The bottom level represents the Read and Write operations executed by the component DBSs. An unusual characteristic is that local transactions do not appear on the top level.

In [BBG89] it is proven that a combined scheduler for an \( n \)-level system, consisting of \( n - 1 \) order preserving conflict based schedulers, used between the pairs of adjacent levels, is correct. This supports the observation made in Example 2 that it is insufficient to guarantee SR on only the bottom layer, where a layer is a pair of adjacent levels. A scheduler for the top layer is also required. The bottom layer scheduler is composed of the schedulers of the component DBSs and its operators are the Reads and Writes issued by the (sub)transactions submitted to the various component DBSs.
3.2.3 A Classification of Scheduling Mechanisms

The top layer scheduler must ensure that all the schedulers at the various component DBSs agree on a serialization order for every global transaction. Concurrency control mechanisms may be classified based on the serialization order they produce [BG81, Wei89]. Dynamic atomic algorithms, such as Strict 2PL, ensure that transactions are SR in the order in which they commit. Static atomic algorithms, such as TSO, ensure that transactions are SR in timestamp order. Some algorithms represent a combination of the dynamic and static algorithms [Wei89].

It is easier to build a top layer scheduler if all component DBSs use a static atomic algorithm since the serialization order can be anticipated by the MDBS from the order in which subtransactions are submitted to the DBS. Most component DBSs use 2PL, a dynamic atomic algorithm, which implies that the MDBS must wait until after subtransactions commit to be able to enforce a serialization order. If executions are not strict, the MDBS will have to wait for a period after commitment until it is certain that the component DBS could not serialize another transaction before it [BST89].

3.2.4 Subtransaction Conflicts

First we must define how subtransactions conflict in the top layer as has been done for Reads and Writes in the bottom layer. It is difficult for the MDBS to determine whether and how subtransactions conflict. Assuming executions are strict, if $G_{iA}$ was committed before $G_{jA}$ ($i \neq j$), then the MDBS can deduce that $G_{iA}$ preceded and could have conflicted with $G_{jA}$. To be sure that two subtransactions did not conflict they must be in a precommit state at the same time. In order to guarantee that an execution is recoverable we assume that all subtransactions that issue Writes to a component DBS must participate in the ACP. Holding locks may result in deadlock. For reasons of simplicity we will assume that a timeout mechanism is used.

In Section 2 three different types of schedulers were mentioned. For each we can build an equivalent top layer scheduler. At the top layer, the operations are the subtransactions submitted to the component DBSs by global transactions.

3.2.5 Top Layer Serialization Graph Testing Scheduler

The Top Layer Serialization Graph Testing (TLSGT) scheduler maintains a type of SG which includes active transactions. This graph is
called the Stored Top Layer Serialization Graph (STLSG). An edge is added to an STLSG if any two operations (i.e. subtransactions) of different global transactions can conflict at a component DBS. Because we are not sure how subtransactions will be serialized before an execution, all possible orders must be considered. Thus for example if $G_{iA}$ and $G_{jA}$ are executed together then the two possibilities $G_i \rightarrow G_j$ and $G_i \leftarrow G_j$ must be considered. Thus several STLSGs will have to be maintained, all of which must be acyclic. The STLSGs for the execution in Example 2 are $G_1 \rightarrow G_2$, $G_1 \leftarrow G_2$ and $G_1 \leftarrow G_2$. Clearly, in the worst case, there can be an exponential number of STLSGs.

When it becomes known how subtransactions were serialized, some edges in the STLSG may be deleted, which will enable other subtransactions to be scheduled. If, for all component DBS $A$ where both $G_i$ and $G_j$ submit subtransactions, either $G_{iA}$ commits before $G_{jA}$ or both are in a precommit state, then $G_i$ is assumed to precede and conflict with $G_j$ and the edge $G_i \rightarrow G_j$ can be deleted. Clearly, if at all component DBSs $A$ $G_{iA}$ and $G_{jA}$ are in a precommit state, then both edges between $G_i$ and $G_j$ can be deleted. These deletions apply to all STLSGs and may result in some STLSGs becoming redundant. If, in Example 2, $G_{1A}$ and $G_{1B}$ had committed before $G_{2A}$ and $G_{2B}$ respectively then the edges in the STLSGs from $G_2$ to $G_1$ can be deleted. The only STLSG that remains is $G_1 \rightarrow G_2$.

Even after deleting edges between $G_i$ and $G_j$ the transactions may still be involved in a cycle of the form $G_i \rightarrow \ldots G_j \rightarrow \ldots G_i$, requiring that a transaction be aborted. If $G_i$ has committed and has no incoming edges then this node and all associated edges may be deleted because it can never be involved in a cycle again.

The ADDS MDBS scheduler [BST87, BS88, BST89] is a variant of a pessimistic TLSGT scheduler with an aggressive scheduling policy [Dea91]. The ADDS scheduler is centralized. A single undirected graph, called a transaction graph is maintained, rather than possibly many STLSGs. In addition to guaranteeing serializability an acyclic transaction graph ensures that deadlock is avoided.

Since one may expect conflicts and deadlock to be rare a more optimistic approach would seem more appropriate. An optimistic scheduler permits subtransactions to execute that will create a cycle in the STLSG. This is done in the hope that the cycle forming edges will be deleted later. The cycles in the STLSG are only permitted while the transactions involved in the cycle can be aborted. The STLSG for all committed transactions may never have cycles. If the cycle forming edge cannot be deleted then a transaction must be aborted to break the cycle.

### 3.2.6 Top Layer Two-Phase Locking Scheduler

Using Top Layer Two-Phase Locking (TL2PL), the MDBS first requests a lock for a component DBS, before a subtransaction is executed. The locks are exclusive. The lock on component DBSs where subtransactions have precommitted or where commitment can be delayed, can be released once all subtransaction locks have been acquired. If subtransaction locks are released early then the component DBS lock must be held until the global transaction terminates. The locking policy must be two-phase, meaning that locks cannot be acquired after they have been released. This would appear to be contradicted by the release of the lock of a precommitted subtransaction. The locks are held, just the granularity has changed from a lock on all data items at a component DBS to those held by the subtransaction.

In the hope that the subtransactions will not conflict, an optimistic TL2PL scheduler can be developed that allows more than one subtransaction to be submitted to a component DBS, if the subtransaction can precommit or control when the locks will be released. This mechanism is the distributed 2PL if all component DBSs support a Precommit command. At component DBSs where the MDBS can only execute read-only subtransactions, because the DBS does not support a Precommit command, the lock on component DBSs must remain exclusive and cannot be released until the global transaction terminates.
3.2.7 Top Layer Timestamp Ordering Scheduler

A Top Layer Timestamp Ordering (TLTSO) scheduler executes subtransactions in timestamp order at each component DBS. The MDBS maintains a global clock. Global transactions submitted are given a unique ever-increasing timestamp which is attached to each subtransaction. If the TLTSO scheduler receives a subtransaction with a timestamp greater than that of the last subtransaction scheduled, the transaction must be aborted; otherwise the subtransaction can be submitted to the component DBS. By executing subtransactions in timestamp order deadlock is avoided.

A conservative TLTSO scheduler waits for an operation to be received from every site before submitting the operation to the component DBS. Thus no global transactions will be aborted due to a subtransaction arriving late. If a site has no subtransaction to send, a null subtransaction is sent.

An optimistic TLTSO scheduler permits multiple subtransactions to execute at a component DBS in the hope that they will precommit in timestamp order. If this does not happen the global transaction must be aborted.

4 Weakness Correctness Criteria

Enforcing serializability in an MDBS is expensive, restrictive and difficult without making assumptions about the component DBSs. For many MDBS applications it may be unnecessary to enforce serializability. A number of mechanisms are considered that are less restrictive in that they allow executions that would not be allowed by the schedulers discussed so far. These mechanisms sacrifice either some correctness or generality. In [KS88] three important mechanisms for enhancing concurrency that are not part of the traditional model are considered: multiple versions, nested transactions and explicit consistency predicates.

First we discuss how multiple versions may be exploited in an MDBS. Next three mechanisms, top layer state serializability, sagas and quasi serializability, are considered that view transactions as being multi-leveled. The quasi serializability mechanism also uses explicit consistency predicates.

4.1 Multiple Versions

Multiple transactions may be executed concurrently at the same component DBS but these transactions may delay other transaction executions. Various proposals for the more efficient processing of read-only transactions in distributed DBSs have been made [GW82, AS89]. Many of these algorithms have relied on multiple versions of data items being maintained, with the read-only transactions being able to read the older versions. This may be implemented in an MDBS if the component DBS use multiple versions. Unfortunately multiple versions cannot easily be exploited in an MDBS to improve concurrency, since transactions cannot select which version of a data item they wish to read without possible sacrificing a large amount of stable storage.

4.2 Top Layer State Serializability

A state SR history satisfies the following [HP86]:

1. The history that includes the operations of all committed transactions leaves the database in a consistent state.

2. Each transaction sees a consistent database.

3. No Writes are lost.

A state SR history is not necessarily SR. For example, \( w_1[x] \) \( r_2[z] \) \( r_2[y] \) \( w_1[y] \) is state SR but not SR. A characteristic of state SR histories is if all operations of read-only transactions are deleted, then the history is SR. For the history above, deleting operations of the read-only transaction produces \( w_1[x] \) \( w_1[y] \), which is SR.

In many applications the type of anomaly in Example 2 is of little consequence. Permitting such executions requires no MDBS scheduling and thus can offer greatly improved performance. The majority of MDBSs currently adopt this approach by limiting support to read-only transactions. Only the component DBSs are able
to execute update transactions. The physical database state remains consistent, although the global transactions may reflect different views of its state. Using the multi-level model we can now recognize that the correctness criterion adopted in these MDBSs is top layer state serializability. Clearly deleting all read-only transaction operations from an MDBS history would make all MDBS histories trivially SR since there would be no global transactions left.

4.3 Sagas

The concept of a saga was proposed for situations where it is either impractical or not possible for a single transaction to execute atomically [GS87]. A saga is a transaction that is executed as a sequence of subtransactions that can be interleaved with other transactions. Either all the subtransactions that comprise the saga are executed successfully or compensating transactions are executed that remove the effects of the aborted saga. Of course other transactions may observe an inconsistent database state and the compensating transactions may not always be able to restore the database to a previously consistent state, but it is assumed that these cases are either rare or of little consequence. This fits into the multi-level nested model, with the saga transactions, the DBS transactions and DBS operations forming the three levels. The implementation of sagas does not require a top layer scheduler.

The saga concept can offer significant improvements in performance and it can be implemented using existing systems [GS87, DHL90]. It would appear that a saga mechanism can be implemented in an MDBS and will enable the component DBSs to function much more autonomously. An example of such an implementation is the semi-transaction concept in [EHHK88]. Here semi-transactions are used in developing an MDBS that integrates worldwide distributed autonomous databases for transnational accounting applications. This approach grew from a strong demand by the banks involved to preserve autonomy of their databases which is not possible if the traditional distributed DBS concepts are adopted.

A transaction can be considered a saga when a sequence of relatively independent steps can be identified where each step does not have to observe the same consistent database state. To amend partial executions of a saga $S_i$, an associated compensating transaction $C_i$ is required. The compensating transaction undoes, from a semantic point of view, the committed effects of $S_i$ but does not necessarily return the data items changed to the same state that existed before $S_i$ executed. This is analogous to the recovery mechanisms used in the multi-level systems discussed in Section 2.3. For example, to cancel a flight reservation a compensating transaction may decrement the number of reserved seats. It would in general be incorrect to return the number of booked seats to its original value since other reservations and cancellations may have been made. However, in the mean time another client may have been refused a seat because all seats were booked, while a subsequent client may have got a seat. This could be seen as unfair.

4.4 Quasi Serializability

Quasi serializability is a concurrency control correctness criterion for a top layer scheduling mechanism that is weaker than serializability [ED91]. As with our model, the definition of a quasi SR history is expressed in terms of global transactions only. However, this is done at the operations level rather than the subtransaction level.

A set of local histories is quasi serial if there exists a total order of global transactions such that for every $G_i$ and $G_j$, where $G_i$ precedes $G_j$ in the total ordering, $o_i$ precedes $o_j$ in a component DBS history for all $o_i$ of $G_i$, $o_j$ of $G_j$, and all component DBS histories [ED91]. The symbols $o_i$ and $o_j$ denote either a Read or Write operation issued by a subtransaction of $G_i$ and $G_j$ respectively. A history is quasi SR if it is equivalent to a quasi serial history.

Example 3

Reconsider the histories in Figure 1. The operations are executed in the order indicated by the arrows. Although the execution of lo-
cal and global transactions is not SR, it is quasi SR with $G_1$ preceding $G_2$ in the total ordering. At $A$, $r_{1A}[z_A]$ clearly precedes $r_{2A}[y_A]$. A possible execution order at $B$ is $r_4[z_2] r_4[w_2] r_1B[u_B] r_2B[z_B] w_4[u_B]$. Here $r_1B[u_B]$ precedes $r_2B[z_B]$, even though $G_2B$ is serialized before $G_1B$.

However, if in Example 2 the write operations of $L_4$ were executed as in Figure 2, then no equivalent quasi serial execution can be found, where $r_1B[u_B]$ precedes $r_2B[z_B]$. Note that the serialization order is the same as that in Figure 1. It is unclear how an MDBS scheduler could differentiate between the execution in Figure 1 and that in Figure 2.

The type of undesirable interactions allowed by a quasi SR execution can be prevented by controlling information flow and imposing the following restriction on the type of global update transaction that can be executed [Bre90]: no data dependencies may exist between data items stored in different component DBSs. This is a type of consistency predicate. Thus for example, a global transaction that adds the value of one account to a second account by executing Write(acc.2 $\leftarrow$ acc.2 + acc.1), can execute only if both accounts are stored at the same component DBS. However, a transaction with the operations Write(acc.3 $\leftarrow$ $200$) and Write(acc.1 $\leftarrow$ acc.1 $-$ $50$) can execute regardless of the component DBSs at which the accounts are stored since there is no dependency between acc.1 and acc.3.

5 Other Correctness Issues

We have thus far limited the discussion to concurrency control issues. There are additional issues that must be considered when choosing a scheduling mechanism for an MDBS that are much more complex than those for conventional DBSs. First we mention some of the problems in developing general solutions to managing replicated data and enforcing semantic integrity. Next some of the compromise solutions and their implications are discussed.

5.1 Problems Integrating Component DBSs

Since both local and global transactions execute Reads on the shared data, schedulers that support high availability of replicated data and guarantee serializability, cannot be implemented in an MDBS. The primary copy, write-all-available and voting based approaches cannot easily be implemented correctly, without also requiring that all transactions that read the replicated data be executed by the MDBS, which violates the autonomy of component DBSs. The only approach that can be implemented correctly is the write-all approach [DL87, Dea91]. A problem with the write-all approach is that update transactions cannot execute immediately if all copies of the replicated data items being updated are not available [BG87].

A second consequence of both local and global operations executing on the shared data is that global semantic integrity constraints cannot be enforced [Lar89]. This is because there can be no guarantee that the integrity constraints enforced by the MDBS and the component DBSs are equivalent.

In addition to the above, there are numerous additional problems in providing integrated functions including creating global schemas and enforcing security.

5.2 Seeking a Compromise

Because of the above problems, together with those of ensuring global deadlock freedom, serializability and recoverability, correctness criteria weaker than serializability are easily justified in an MDBS. The ideal MDBS should not attempt to provide the complete transparency and functionality, as for example in the list proposed by Stonebraker, for an ideal distributed DBS [Sto89]. Rather a compromise is needed between degrees of correctness, autonomy, data sharing and transparency that can be efficiently implemented and provides the type of functionality needed by users. One must be careful in seeking a compromise, that "correct" and "less correct" mechanisms are used together, since the
consistency of the data is related to "least correct" mechanism used.

5.3 Compromise Solutions

Using the multi-level model and the knowledge of which transactions are precommitted and those still executing at the same component DBSs we can build a Top Layer Wait-For Graph (TLWFG) to detect possible global deadlock situations (c.f. Section 2.2.1). The TLWFG for the execution in Example 1 is $G_1 \subseteq G_2$. A timeout mechanism is much less pessimistic than a top TLWFG testing mechanism or similar mechanisms mentioned in Section 3.1. Timeout mechanisms have the disadvantage that a knowledgeable person must select a suitable timeout period and transactions may be aborted although there is no deadlock. A compromise solution is to check the TLWFG after a timeout period has expired to determine if global deadlock is possible [BST90]. A decision may then be made by the MDBS to either abort a global transaction or reset the timeout period. This can make choosing a suitable timeout period less difficult.

Various MDBSs proposed the use of an "optimistic commit protocol" that uses a top layer scheduler mechanism to ensure that maximally one subtransaction can update a component DBS's database [BST87, TTC*90]. This can reduce the risk of the database becoming corrupted. It operates under the assumption that all subtransactions can commit; if one aborts, it must be re-executed until it commits. The component DBSs thus need not support a Precommit command. This mechanism has the disadvantage that it cannot tolerate the failure of component DBSs, since a failure may make it impossible for the MDBS to either determine or control which transactions are aborted or committed by a component DBS. It seems inappropriate to use such a "less correct" mechanism together with one that enforces SR read transactions as was proposed for ADDS [BST87].

One restriction that will enable the enforcement of global integrity constraints is to restrict the component DBSs to updating certain data items; while other data items can only be updated by the MDBS [ED91]. Integrity constraints can then be enforced on these disjoint sets of data items, since the problem of ensuring the constraints enforced by the component DBSs and the MDBS are equivalent, is now eliminated. If multiple global schemas can exist, supporting global integrity may be inappropriate [DL87]. It is difficult to support a single global schema if a flexible mechanism for updating replicated data is not available, since updating it then becomes problematic.

Even if the correctness criteria are weakened, it is very difficult to implement schedulers that offer high availability for replicated data. The primary copy approach represents a compromise which offers users the choice between reading the primary copy and a possibly out-of-date non-primary copy [DAOT85]. Making other restrictions on distributed reads, a more flexible primary copy mechanism can be implemented [GA87].
6 NRDStar MDBS Project

NRDStar is an MDBS currently being developed in the Department of Computer Science at Stellenbosch University. At present support is limited to read-only transactions. The results of the research reported here are being applied to extend support to updates. It is our intention to implement several mechanisms, including distributed 2PL, saga and primary copy mechanisms. When NRDStar is loaded into a new environment, one or more of these mechanisms may be installed, depending on the user’s application. In addition, NRDStar may either be installed with either only one or multiple global schemas. This will allow NRDStar to be used for many applications which have widely varying correctness criteria.

7 Conclusion

The multi-level nested transaction model was successfully used to model MDBS transaction executions. The model enabled a number of new MDBS schedulers, called top layer schedulers, to be developed. This was achieved using existing theory and concepts; we only had to define how subtransactions conflict. These schedulers can be proven correct using the theory in [BBG89]. Unfortunately, with the exception of distributed 2PL, none of these scheduling mechanisms appears to be very practical. This supports the comment in [SL90] that it is unlikely that there exists a theoretically elegant solution that guarantees serializability without sacrificing performance and availability.

Several weaker correctness criteria and restrictions on processing have been proposed. The majority of MDBSs currently limit support to read-only transactions and perform no concurrency control. Applying the multi-level model we were able to recognize the correctness criterion used in these systems as being top layer state serializability. Update transactions are supported in the saga mechanism and mechanisms that enforce quasi serializability. A saga is a nested transaction executing in a system without a top layer scheduling mechanism. Limited recover-

ability is achieved using compensating transactions. Quasi serializability is a weak top layer correctness criterion that is correct as long as there are no value dependencies across component DBSs.

In a MDBS, support for recovery, global deadlock resolution, replicated data management, global semantic integrity control etc. may also be needed. However, as with concurrency control it does no appear that efficient general solutions are likely to be found. More restrictive compromise solutions have to be sought that are compatible with one another, do not violate the autonomy of component DBSs, can be implemented efficiently and that will solve users’ problems. The solution space has not been fully explored, which will require continued research [SYE90]. We feel our model can serve as a useful basis for such research.

8 Acknowledgements

We would like to thank Martin Rennhackkamp and the anonymous referees for their constructive comments which led to an improvement in the clarity of this paper.

References


Extending Local Recovery Techniques for Distributed Databases

H L Viktor and M H Rennhackkamp

Abstract

The most frequently used local database system failure recovery techniques are logging, shadowing and differential files. In a distributed database these local system failure recovery techniques may be utilized for recovery from a single site failure. However, these techniques need to be extended to facilitate continued distributed executions. Various extended local system failure recovery techniques are presented. The results of a comparison of these techniques are shown. It is concluded the deferred data item logging technique proves to be the best for the system under consideration.

Keywords: Database management, distributed database system, failures, recovery techniques, recovery.
Computing Reviews Categories: H.2.2, H.2.4, H.2.7

1 Introduction

During a local system failure the contents of main memory is lost and processing is terminated at the site at which the failure occurred. This could result in an inconsistent local database, since logically completed updates may not be reflected in the physical database files. At restart, the results of such locally committed, unpropagated executions should be propagated. Operational sites can continue execution during the failure of another site. This increases the complexity of distributed database recovery. At restart, the outcome of partially completed distributed executions should be determined. In addition, copies of replicated data items may be outdated. These copies should be updated to reflect the new values.

The recovery management function provides the facilities to preserve the consistency of the database in the event of a failure. In a distributed environment these facilities include techniques to consistently recover the local database, bearing in mind distributed recovery aspects. Various local system failure recovery techniques for centralized databases exist. The design and implementation of these techniques are well understood. In a distributed database these techniques may be extended to facilitate for the recovery from a site failure caused by a local system failure. Few authors have addressed distributed database recovery utilizing local recovery techniques. The results of such an extension are presented.

In the next section, a distributed transaction execution model is presented. This study forms part of the NRDNIX DDBMS, which is currently being developed at the University of Stellenbosch. An overview of this system is given. This is followed by a description of widely used local system recovery techniques. The extensions to these techniques, as required by the NRDNIX DDBMS, are discussed. Lastly, results of a comparative modelling of these extended techniques are presented.

2 Distributed Transactions

A transaction performs operations on the data items in a database. The transaction execution includes the reading, possible modification and the subsequent writing of data items. A sequence of operations constituting a transaction are delimited by BEGIN-TRANSACTION and
COMMIT or ABORT keywords. It is a sequence of actions considered as an atomic logical unit of work. If the transaction completes successfully, it is committed. Otherwise, it is aborted and no effects of the transaction remains in the database.

In a distributed environment transactions perform operations on data scattered across sites of a network. A distributed transaction is executed by processes at these sites. These processes, called subtransactions, are able to communicate and cooperate with each other.

The actions performed by a transaction should be indivisible. Either all of a transaction’s actions should be properly reflected in the database, or none. This indivisibility requirement is met by a transaction satisfying the following characteristics, also referred to as the ACIDS-principle:

1. Atomicity. Acceptance of the transaction should be a guarantee that it will be run exactly once, any update executed only once and its results produced exactly once. The atomicity of distributed transactions are guaranteed by utilizing an atomic commit protocol.

2. Consistency. Whenever a transaction executes on a database state that is initially consistent, it must leave the database in a consistent state after it terminates. A successful distributed transaction results from the execution and completion of successful subtransactions.

3. Isolation. The events within the transaction should be hidden from others. From the distributed transaction definition it follows that this property is satisfied.

4. Durability. Once a transaction has successfully completed execution, the system should guarantee that the results of its operations will never be lost, except in the event of a catastrophe.

Once a distributed transaction has committed its results should be guaranteed to be permanent, even in the event of a failure. The results of subtransaction execution cannot be ensured since a successful subtransaction is aborted if the coordinator issues a global abort.

5. Serializability. In most systems transactions are allowed to execute concurrently. The concurrent execution of transactions should be guaranteed to be serializable. An interleaved execution of transactions is serializable if it produces the same output and has the same effect on the database as some serial execution of the same transaction. The activity of guaranteeing transactions’ serializability is called concurrency control. In a distributed database, the global as well as the local concurrency must be maintained.

A transaction which satisfies the ACIDS-principle is said to be successful. A database is consistent if and only if it contains the results of successful transactions [11].

The indivisibility of a distributed transaction is enforced by employing an atomic commit protocol. The basic idea of an atomic commit protocol is to determine a unique decision for all participants with respect to committing or aborting a transaction. The most widely used commitment protocol is the two phase commit (2PC) protocol. The protocol consists of two phases. During the first phase, the coordinator decides to either commit or abort the distributed transaction. The second phase is used to implement this decision. If a participant is unable to locally commit the subtransaction, then all participants must locally abort [7].

The 2PC protocol has been extended to eliminate unnecessary blocking during site failures [22]. Here, an additional phase 2a is introduced. The participants do not directly commit the transaction during the second phase of commitment. Instead, they reach a new prepared-to-commit or prepared-to-abort state. This modified 2PC protocol is utilized in the distributed database management system (DDBMS) under consideration.
3 NRDNIEX DDBMS

In this section an overview of the NRDNIEX system is given. The design areas which are of importance when discussing recovery related aspects are highlighted.

The NRDNIEX DDBMS is implemented as a number of processes, called Managers, on XENIX, a version of UNIX System V [17].

Presentation Manager

The aim of the presentation manager is to form an efficient user interface to the DDBMS. It accepts user requests and transforms them to optimized internal commands according to information obtained from the data dictionary. The communication kernel is activated to execute the internal commands. The presentation manager receives the results of these executions and presents them to the users.

A distributed transaction consists of one or more sessions. The internal commands which forms part of a distributed transaction are decomposed into sessions. Each group of related single relation operations, each binary join operation and the final project operation constitute a session. A session is considered a unit and a session sequence number is assigned to it. Concurrent split processing is utilized. The local executions of a distributed databases' sessions are grouped into subtransactions which are concurrently executed at all operational sites in the network.

Data Dictionary

The function of the data dictionary is to manage the central inventory of the metadata, i.e. descriptive data about the data stored in the database. It provides a number of functions for managing, inserting, updating and retrieving metadata from the system relations. The data dictionary maintains relevant information to support the management of replicated data.

Communication Kernel

The communication kernel controls the concurrent execution of distributed transactions. The site at which a transaction is initiated is the coordinator of the transaction. The other sites are referred to as participants.

The communication kernel consists of 5 components.

- The transaction manager identifies and manages the sessions of locally issued transactions. The global execution of the session is initiated by requesting the network manager to broadcast the session to all remote sites. The concurrency controller is requested to schedule the session locally. The results of executed transactions are returned to the presentation manager.

- The slave session manager controls the local execution of remotely issued sessions. It acts as a participant in the transaction of which the session forms part.

- The master session manager controls the global execution of each locally initiated session and thus coordinates the transaction's sessions.

- The concurrency controller schedules the execution of transactions in conservative timestamp order. Sessions received from both local and remote sites are buffered in timestamp ordered lists.

- The recovery manager is responsible for the coordination of all recovery related aspects introduced by the distributed nature of the database. Its task includes the recovery from
site failures as well as network partitions. The recovery manager also controls the global commitment of transactions by utilizing the modified 2PC protocol.

Database Manager

The database manager is responsible for the physical retrieval of data and the physical execution of operations. It consists of the cache and access managers. The cache manager controls the execution of operations on the database. The access manager forms the interface between the cache manager and the XENIX file system. It is primarily responsible for managing records of data files. Accesses to the local database files are attained through the XENIX kernel. The access manager of the local DBMS interacts with the XENIX kernel via standard system calls. All low level operations on the database files are therefore controlled by the XENIX file system, thus abstracting the DBMS from the physical disk accesses.

Network Manager

The network manager provides a reliable error-free communication service to the communication kernel. The ArcNet local area network communication facilities are used. ArcNet supports a modified token passing protocol on a bus architecture. It forms a virtual ring, where each site receiving the token acknowledges it.

A DDBMS component wishing to communicate with another site sends a message via the network protocol manager. A message is divided into fixed length packets, which are sent to the network device driver. The device driver is responsible for the actual communication.

The network protocol manager employs point-to-point communication with packet acknowledgement. In addition, a reliable broadcasting facility is offered when not more than a specified threshold of sites fail. The broadcasting facility is used when a coordinator initiates remote sessions [16].

4 Local System Failure Recovery Techniques

Three basic techniques are widely used for recovery from local system failures, namely logging, shadowing and differential files [1, 12].

4.1 Log Files

An incremental log, also called journal or audit trail, is a representation of the history of transaction execution at a particular site. It records all actions performed on the local database. Data are collected for the sole purpose of recovering invalid data from the local database and supplementing the local database with updates of completed transactions that were not yet reflected in it at the time of failure [11, 18, 2].

Recovery data are written to the log prior to the actual transaction execution. Write ahead logging satisfies the following two rules [8, 11]:

1. Undo information is written to the log file before the corresponding updates are propagated to the materialized local database.

2. Redo information is written to the log before a transaction is committed. The system must be able to ensure the transaction's durability once the redo information has been written.

Logging is usually combined with some form of checkpointing. A checkpoint records the local database state at a particular instance. Checkpoints are used to eliminate unnecessary redo of transactions which have already written their updates to the local database, as well as
to obtain the list of transactions to be undone or redone. At restart, the checkpoint is used to generate undo and redo lists. At a checkpoint all log information is written to nonvolatile storage. All modified local database file blocks are propagated. At the completion of the checkpoint, a checkpoint record is written to the log. The checkpoint record may contain lists of the currently active and recently aborted or committed transactions.

There are two basic approaches to log transaction executions, namely physical and logical logging [11, 18].

4.1.1 Physical Logging

Some part of the physical presentation of modified data is written to the log. Either the state, before or after a change, or the transition causing the change is logged. A before-value log file record the value of modified data before a transaction execution and is used by the undo algorithm. After-value log files, which record the value of data items after updates have occurred, are utilized by the redo algorithm.

Physical Page Logging

The most basic physical logging method uses a page as unit of log information. The before-image and/or after-image of a page, or the difference between these images, is written to the log.

- With state logging, the pages containing the changes executed by a transaction operation are written to the log. The before-images are written to the log file before the corresponding updates are applied to the local database. The after-image of a page is written to the log file before the transaction is committed.

At system restart the undo algorithm uses the before-images of all the modified pages to restore incomplete transactions. The redo algorithm uses the after-image of each page to propagate the results of committed, but as yet unpropagated, transactions.

- With transition logging, the exclusive-or’ed difference between the old and the new page is written to the log before it is propagated to disk. Usually only a small part of data on a page are affected by a change. The xor’ed difference will thus contain long strings of 0’s, which are removed by compression techniques [11]. With compression techniques, savings of 20 to 50 percent are typical for text files.

At system restart the xor is applied to the decompressed pages. By applying the xor difference to the before-image of a page, the after-image are obtained. On the other hand, applying the xor to a after-image will subsequently yield the before-image.

Physical Data Item Logging

The changes to data items are logged, rather than the whole page.

- In the deferred write approach all updates on data-items are recorded on the log, but the actual writes are deferred until the transaction partially commits. The log consists of entries of the form \([T, ts, x, v]\), identifying the value \(v\) that transaction \(T\) wrote into data item \(x\) at time \(ts\), thus reflecting the state and the transition of a data item after modification. When the transaction partially commits, a commit record is written to the log. The actual updating of the records is initiated by applying the log to the physical local database.
At restart, a no-undo, redo algorithm is executed. If the failure occurred after the commit of a transaction was recorded in the log, a redo algorithm updates the database files to include the results of committed unpropagated values. The log entries regarding incomplete or aborted transactions are simply ignored.

- With the immediate write approach all updates are immediately applied to the local database. The write ahead log contains all changes to data items. The log consists of entries of the form \([T, ts, z, o, v]\), identifying the old value \(o\) of data item \(z\) as well as the new value \(v\) that transaction \(T\) wrote at time \(ts\).

At restart, the log information is used to restore the state of the system to a previous consistent state. All committed, but unpropagated transactions are redone by propagating the new value \(v\). Similarly, all relevant active or aborted transactions are undone by propagating the old value \(o\).

4.1.2 Logical Logging

A logical log consists of a sequence of transaction operations executed on the local database. The operations, the arguments they operate on, as well as some control information are logged prior to the execution thereof. Thus the log is abstracted from the physical level. No information regarding the actual accessed data items are maintained.

At restart, committed unpropagated transactions are redone by re-executing the data manipulation language (DML) statements as recorded in the log. While it is possible to remove the effects of aborted transactions by executing an undo algorithm, such an undo algorithm may introduce inconsistencies and should therefore be avoided [21]. The logical logging recovery technique need therefore be combined with another recovery technique which utilizes a no-undo algorithm.

4.2 Shadows

Two copies of the data being updated during transaction execution are kept, the original copy referred to as the shadow and a modified current copy. When a transaction commits, the shadow copy is replaced by the newly updated copy [12, 14, 15]. These two copies exist only during updating; otherwise only a single copy exists which contains the current value.

Shadowing is difficult to implement in systems which allow several transactions to execute concurrently [14]. If several transactions concurrently alter a file, file save or restore is inappropriate because it aborts or commits the updates of all transactions to the file. It is desirable to commit or abort on a per transaction basis.

4.2.1 Shadowing and Logging

Shadow paging is combined with logging [10] to facilitate for concurrent transaction execution. A current and shadow version of a file is maintained. When a shadow page is updated for the first time, a new disk page is assigned to it. Thereafter, when the page is read from or written to disk, the new page is used. The shadow is never updated. Saving the results of a transaction consists of writing to disk all altered relevant pages currently in main memory and freeing superseded shadow pages. At the commitment of a transaction the results are propagated to disk. The results of all transactions which have previously committed will be reflected in the database.

Aborting a transaction is achieved by discarding pages of that file in the buffer pool, freeing all the new disk pages of that transaction and reverting to the shadow page table. At restart all shadowed files are reset to their shadow versions. The log is used to remove the effects of aborted transactions and to restore the effects of committed transactions.
4.2.2 Transaction Orientated Shadowing

Another approach, based on ideas by Lorie [15], is presented by Agrawal [1]. For each relation a shadow page table is maintained. For each transaction an incremental current page table is formed in main memory. When a transaction wishes to commit, the transaction’s current page table is written to a commit list. This acts as a transaction’s precommit record. It ensures that all updated buffer pages corresponding to the transaction are output to disk. The current page table of a transaction is used to update the system shadow page table. The disk address of the new system current page table overwrites the address of the old system shadow table. The system shadow table is discarded and the current system table becomes the new shadow as the transaction commits. Finally, a commit statement is written to the commitlist.

At restart, the current page tables of transactions are discarded to remove the effects of incomplete executions. The commitlist is examined to determine those transactions for which a precommit record appears in the list, but not the commit record. For all such transactions, the shadow system page table is updated using the precommit record.

4.3 Differential Files

All logical files comprise of two physical files: a read-only database file and a read-write differential file [19]. Accessed data items are maintained in the differential file. The differential file is always searched first when data are retrieved, thus obtaining the most recent entry for a given item. Data not found in the differential file are retrieved from the main local database. A hashing scheme is usually implemented to minimize the overhead associated with the determination of the location of an item. The base file remains unchanged until a merging algorithm is executed. During merging the results of committed transactions are moved from the differential file to the local database file [21].

4.3.1 Differential Files and Logging

All updates by concurrent transactions are written to one differential file. The differential file is combined with logging to determine the transaction execution sequence. The transaction identifiers, a unique timestamp as well as information on the transaction’s operations are written to the log.

At restart, the logical database consists of the main database and differential file. In addition, committed transactions which have not yet been written to the differential file are redone by utilizing the logging information.

At a differential file checkpoint, all buffer pages are written to the differential file by utilizing one of the local checkpointing schemes. In addition a new differential file may be opened. The results of active transactions may be moved to a new differential file. The current differential file, which contains only the results of committed transactions, is closed and the new differential file becomes current.

4.3.2 Transaction Orientated Differential File

Each transaction maintains its own transaction differential file which is inaccessible to other transactions. The differential file is decomposed into two files: an append file A and a deletion file D. Each logical file R is considered a view $R = (B \cup A) - D$, where B is the read-only base portion of R. Each transaction is assigned a unique timestamp. The differential file tuples are widened to include an extra timestamp. While a transaction is active, its updates are executed on its local $A_t$ and $D_t$ files that are inaccessible to other transactions. When the transaction commits, the local files are appended to the main differential files $A_g$ and $D_g$. Finally, the timestamp of the committing transaction is written to a commitlist. If a transaction aborts,
its private $A_i$ and $D_i$ files are simply discarded and its timestamp is not appended to the
commitlist.

To recover from a system crash, instead of R, a view of R is used that consists solely of
the tuples whose timestamp fields contain values that appear in the commitlist. The logical
database consists of the main database as well as the results of transactions included in the
commitlist. These committed transactions are contained in the main differential file.

5 Distributed Recovery

The autonomy of sites are of critical importance when designing a DDBMS [7]. A NRDNIX
recovery manager must be able to recover a site to a consistent state autonomously. The previously
introduced local recovery techniques need to be extended to be utilized in the NRDNIX
distributed environment.

The degree with which distributed transaction executions satisfy the ACIDS-principle is used
as metric to evaluate the correctness thereof. The presented extended local system recovery
techniques satisfy the ACIDS-principle. A globally consistent database state is reached at the
successful completion thereof [22].

5.1 Logging

The results of locally initiated distributed transactions and subtransaction executions are logged
utilizing local logging. Commit processing information is also logged. This occurs at all sites
of the network.

At restart, a previous consistent local database state is obtained by consulting a previous
checkpoint. In distributed databases, a global checkpoint refers to a set of saved local states
of sites. Such a set of local checkpoints is consistent if it forms a globally consistent state
[5, 9, 13, 22].

The results of subtransactions which have committed since this last checkpoint are redone.
The results of distributed executions which occurred during the failure are obtained from the logs
of operational sites via the communication network. The local redo algorithm is applied to the
remote log information. First, subtransactions of partially completed distributed transactions
are completed. Second, the results of newly submitted transactions are made part of the
database.

The remote information may be obtained in one of two ways. One or more sites may
be elected as the source. Alternatively the recovery information may be received from all
operational sites. In NRDNIX, a transaction is concurrently executed at all sites of the database.
It is therefore sufficient to request the logged information from only one site. Usually the log
of the site which acted as token during the failure is utilized.

5.2 Shadowing and Logging

Locally initiated distributed transactions and subtransactions execute on current copies of the
local database files. These executions are logged prior to their execution. Commit processing
information is also logged as part of the local log. Local checkpointing is implicitly executed
each time a subtransaction commits during the last phase of the commit protocol.

At restart, a previous consistent database state is obtained by starting from the local shadow
file. The local log is consulted and the results of all locally committed, but as yet unpropagated,
transaction executions are executed on this copy. The recovering site obtains the results of
partially completed transactions from the commit log of the token site. These transactions are
locally completed. The log at the token site is consulted to obtain the results of newly submitted
transactions. These results are made part of the local log by execution a redo algorithm.
5.3 Transaction Orientated Shadowing

A separate current file is created for each subtransaction of a distributed transaction. The current page table is written to a commitlist during the last phase of the commit protocol. Once this has been done, it is ensured that the results of the subtransaction will survive a site failure. Finally, a commit record is written to the commitlist.

At restart, the local commitlist is examined to determine those subtransactions for which a precommit record appears in the list but not a commit record. The shadow page table is updated using the precommit record. The commitlist is obtained from an operational site. The remote shadow file information of all newly submitted transactions are made part of the local database.

5.4 Differential Files and Logging

One main differential file is utilized. The results of subtransactions of locally as well as remotely issued distributed transactions are maintained therein.

At restart, the main database together with the committed transactions recorded in the differential file reflect a consistent state. The results of locally committed, unpropagated transaction executions may be found at two places: in the differential file or in the log. The results of committed logged transactions which are not contained in the differential file are made part of the database by executing a redo algorithm.

The results of recently completed distributed transactions are located in the differential files of operational sites and are located in their logs. One of two approaches may be utilized. In the first, the remote log information is obtained and the transactions are explicitly re-executed. In the second, both the remote differential file and log information are utilized. The differential file records corresponding to the committed transactions are selected and are copied to the local database files.

5.5 Transaction Orientated Differential Files

A logical transaction differential file is utilized by the subtransactions at each participant in a distributed transaction. Physically this file consists of two files: a deletion file \( D_I \) and an add file \( A_I \). When a subtransaction is locally committed, this local differential file is appended to a main differential file \( A_g \) and \( D_g \). The timestamp thereof is written to a commitlist. The commit information is maintained in the commitlist.

At restart, a previous consistent database state is constructed. This state consists of the main database file as well as the records in the main differential file for which an entry is the commitlist has been recorded. In addition, the information of transactions committed during the failure are made part of the local database. This information is obtained from operational sites. The relevant parts of remote main differential files are received via the network. It is appended to the main differential file at the recovering site.

6 Comparison of Techniques

A model to uniformly model the overhead associated with the various recovery techniques was developed [22]. The various techniques are uniformly modelled against the same set of transactions, number of sites and distributed database specifications. These specifications are directly derived from the way processing is performed in NRDNI.X.

Figure 1 shows the parameters which are used in the modelling. The distributed database consists of 5 sites. For this example the maximum number of records at a site is taken as 400. A total of 19 distributed transactions are concurrently executed at the 5 sites in the distributed
database. These distributed transactions are in different stages of execution when the failure occurs [22].

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sites</td>
<td>5 sites</td>
</tr>
<tr>
<td>Maximum Records per Site</td>
<td>400 records</td>
</tr>
<tr>
<td>Total Distributed Transactions (DTs)</td>
<td>19 sites</td>
</tr>
<tr>
<td>Completed and Committed DTs</td>
<td>5 sites</td>
</tr>
<tr>
<td>Completed and Aborted DTs</td>
<td>5 sites</td>
</tr>
<tr>
<td>Restarted and Committed DTs</td>
<td>5 sites</td>
</tr>
<tr>
<td>Restarted and Aborted DTs</td>
<td>4 sites</td>
</tr>
</tbody>
</table>

Figure 1: Modelling Parameters

The disk storage, message and I/O overheads on the distributed transactions, imposed by the extended recovery techniques, are determined.

**Disk Storage Overhead**

Disk storage overhead is incurred by the redundant recovery data which are accumulated and maintained during normal processing [10, 12, 20]. The amount of additional disk storage overhead is determined by the number of pages written to disk. The page length is considered to be 1024 bytes, corresponding to the size of a XENIX page.

**Message Overhead**

The additional message traffic [6] imposed by the recovery techniques are taken as metric when discussing message overhead, since it will directly influence the transmission delay and thus the overall transaction execution. In the NRDNIK system, the fixed cost associated with each message dominates the overall transmission cost. The message transmission cost is therefore not much influenced by the variable cost caused by the message length [16]. The number of additional messages is easily determined and is sufficient to model the message overhead [6].

**I/O Overhead**

The additional I/O operations are incurred when accessing and propagating recovery related data to disk and when executing I/O operations during recovery. The movement of data is performed by executing a XENIX read or write system call. The number of additional I/O cycles are used as the unit of modelling when discussing I/O overhead [1, 10, 12].

The CPU overhead imposed by the recovery techniques are implicitly included in the I/O costs. The reason therefore is threefold. Firstly, the ability of the system to exploit the capacity of faster CPUs is directly influenced by the amount of I/O operations [18]. Secondly, a large amount of the CPU overhead is incurred by the setting up of the I/O operations [12]. Thirdly, the I/O and communication costs are shown to dominate the processing costs [3].

**7 Results and Evaluation**

The results of the comparison are shown in figures 2 and 3. These values reflect the overhead during normal and recovery processing [4, 22]. The normal processing overhead is taken as the cost incurred during a fixed time interval T. This overhead is calculated by considering all sites in the distributed database. The resulting values reflect the overhead at the most
expensive site. The recovery processing overhead is calculated at the recovering site. It consists of the reprocessing overhead required to recover the set of distributed transactions to a globally consistent state.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Disk bytes</th>
<th>I/O cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page State Log</td>
<td>69 836</td>
<td>102</td>
</tr>
<tr>
<td>Page Transition Log</td>
<td>28 016</td>
<td>68</td>
</tr>
<tr>
<td>Deferred Data Item Log</td>
<td>986</td>
<td>136</td>
</tr>
<tr>
<td>Immediate Data Item Log</td>
<td>1054</td>
<td>102</td>
</tr>
<tr>
<td>Deferred Data Item Log &amp; Shadows</td>
<td>3366</td>
<td>238</td>
</tr>
<tr>
<td>Immediate Data Item Log &amp; Shadows</td>
<td>3434</td>
<td>204</td>
</tr>
<tr>
<td>Logical Log &amp; Shadows</td>
<td>3672</td>
<td>153</td>
</tr>
<tr>
<td>Transaction Orientated Shadows</td>
<td>238</td>
<td>255</td>
</tr>
<tr>
<td>Transaction Orientated Diff. Files</td>
<td>34 918</td>
<td>221</td>
</tr>
<tr>
<td>Immediate Data Item Log &amp; Diff. Files</td>
<td>36 040</td>
<td>204</td>
</tr>
<tr>
<td>Deferred Data Item &amp; Diff. Files</td>
<td>35 972</td>
<td>238</td>
</tr>
<tr>
<td>Logical Log &amp; Diff. Files</td>
<td>36 244</td>
<td>153</td>
</tr>
</tbody>
</table>

Figure 2: Normal Processing Overhead per site.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Disk bytes</th>
<th>I/O cycles</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page State Log</td>
<td>32 832</td>
<td>136</td>
<td>32</td>
</tr>
<tr>
<td>Page Transition Log</td>
<td>13 129</td>
<td>136</td>
<td>13</td>
</tr>
<tr>
<td>Deferred Data Item Log</td>
<td>448</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td>Immediate Data Item Log</td>
<td>480</td>
<td>136</td>
<td>1</td>
</tr>
<tr>
<td>Deferred Data Item Log &amp; Shadows</td>
<td>1506</td>
<td>236</td>
<td>2</td>
</tr>
<tr>
<td>Immediate Data Item Log &amp; Shadows</td>
<td>1440</td>
<td>142</td>
<td>2</td>
</tr>
<tr>
<td>Logical Log &amp; Shadows</td>
<td>1552</td>
<td>36</td>
<td>2</td>
</tr>
<tr>
<td>Transaction Orientated Shadows</td>
<td>204</td>
<td>236</td>
<td>1</td>
</tr>
<tr>
<td>Transaction Orientated Diff. Files</td>
<td>16 432</td>
<td>40</td>
<td>16</td>
</tr>
<tr>
<td>Immediate Data Item Log &amp; Diff. Files</td>
<td>16 928</td>
<td>136</td>
<td>17</td>
</tr>
<tr>
<td>Deferred Data Item &amp; Diff. Files</td>
<td>16 896</td>
<td>68</td>
<td>17</td>
</tr>
<tr>
<td>Logical Log &amp; Diff. Files</td>
<td>17 040</td>
<td>36</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure 3: Recovery processing overhead

Normal Processing Overhead Evaluation

During normal processing, message overhead is incurred by the execution of the commit protocol. This overhead is equal for all techniques and is therefore not shown.

The transaction orientated shadowing technique yields the best disk storage overhead, but the I/O cost is high. The physical page transition logging technique yields the best I/O overhead results, with a high disk storage overhead. Both the physical data item logging approaches yields both low disk storage and I/O costs, with the deferred write approach slightly better than the immediate write approach. Data item logging therefore seems the most suitable method for the system under consideration.
Recovery Processing Overhead Evaluation

The transaction orientated shadowing technique yields the best disk storage overhead and message overhead, but the I/O cost is high. Logical logging combined with the shadowing techniques yields the best I/O overhead with a moderately low disk storage and message overhead. Both the physical data item logging approaches yields low disk storage, I/O and message costs, with the deferred write approach slightly better than the immediate write approach. Physical data item logging therefore seems the most suitable method for the system under consideration.

8 Conclusion and Extensions

Local system failure recovery techniques are widely utilized in centralized databases. These techniques need to be extended for a distributed environment. The extension of local system recovery techniques is useful since the distributed database management system utilizes existing and well understood techniques. The autonomy of sites are preserved since the recovery management function is localized.

An overview of the extended local system failure recovery techniques was given. The results of a comparative modelling of these techniques were presented. The modelling is aimed at a specific distributed database environment. The deferred data item logging technique proves the best technique for the specific system, both during normal and recovery processing.

The paper concerns the failure of one site due to local system failures. Multiple site failures and network partitions were not discussed. An extension to this work presents techniques and modelling to incorporate these types of failures [22]. In addition, the model should also be applied to other distributed database environments.

References


Analysing Routing Strategies in Sporadic Networks

Stuart Melville
Department of Computer Science
University of Natal, Durban

Abstract

Communications networks with sporadically available links pose particular problems in terms of delay analysis. This paper presents an approach to determining the effectiveness of various routing strategies when applied to such networks, the strategies considered being minimum-hop routing, shortest-path routing, and flood routing. The implementation of a simulator package for sporadic networks is discussed, with particular attention to a branch-and-bound algorithm devised for the efficient simulation of flood routing. Meteor-Burst Communications networks are used as primary examples of sporadic networks, and some preliminary results are presented.

Introduction

In most forms of communication networks links between stations, once established, are constantly available. Thus the delay involved in sending a message from one station to another is essentially invariant if factors such as contention, buffer overflow and so forth are excluded from consideration.

Some networks, however, have links which are only sporadically available. Henceforth we shall describe such networks as 'sporadic networks'. A typical example is Meteor-Burst Communications, where signals between stations are scattered off the trails of ionisation created by meteors entering the atmosphere at high speed. While billions of meteor trails are formed every day, a trail must occur in the region of sky illuminated by both the transmit station and the receive
station for the link between them to be viable. Different links will have different mean delays between usable meteor trails, due to such factors as transmitter and receiver power, galactic noise, size of common area of sky, and so on.

Modelling throughput over such sporadic links is not problematic (see [1],[2]) as the channel can merely be perceived as a low data-rate one rather than a collection of 'bursts' at high data rates. This is not valid in terms of delay analysis however (see [3]), as modelling the high data-rate sporadic channel as a low data-rate continuous one would result in gross underestimation of average delay for 'short' messages.

Some useful work has been done in analysing routing strategies for fast-changing networks (see [4]), and there has been a belief that this work might be applicable to sporadic networks ([5]). However there is a fundamental difference between these two types of networks. In the fast-changing network, typically one with mobile stations, the actual stations and established links change over time. In the sporadic network stations and the links between them are constant, but the links are only viable at certain times.

**Routing Strategies Considered.**

There is of course a plethora of potential routing strategies. In this work three of the more basic approaches are considered. These are minimum-hop routing, which seeks to route messages along the path containing the least number of intermediate stations; shortest-path routing, where messages are routed along the path estimated to have the least total delay; and flood routing, where the originating and all intermediate stations forward copies of messages to each of their neighbours which are not on the path the current copy originated from. In flood routing a record is kept of what messages have been forwarded, so duplicate copies arriving later may be ignored.

If there was no variance associated with delay in a sporadic network, and no possibility of channel noise causing more than one attempt to be needed for each send, the time taken for a message using shortest-path routing would be identical to that taken using flooding (barring consider-
ations like contention, buffer storage, etcetera), as the best flood path would of course be the shortest path. However in sporadic networks which do have a non-zero variance of delay between usable links, this is not always the case. While the path with the shortest mean delay can be established, there is no guarantee that this will be the shortest path for a particular send. Thus it is quite possible for the flood routing to find a path for a particular message which is ‘shorter’ (less delay) than the path of least mean delay.

Simplifying Assumptions.

A number of simplifying assumptions are made in order to constrain the problem of comparing different routing strategies. Firstly, it is assumed that there will be no contention within the network. While this may seem an extreme assumption in terms of conventional networks, it is somewhat more tenable for those sporadic networks in which the availability of any given link is independent of the availability of all others (bar the special case of reciprocal links). Certainly collisions will be infrequent is such networks, however there could still be some queuing contention. The circumstances in which this contention would arise, as well as its effects, shall be discussed in detail later in the paper.

Further, it is assumed that all stations have unlimited buffer storage, so no overflow problems can occur, that there is a constant transmission error rate across all links in the network, with a simple ARQ feedback scheme being employed, and that traffic flow between stations is uniform (ie. the network does not have any overly ‘loquacious’ or ‘reticent’ stations).

In addition, it is assumed that all messages are small enough to be sent within the time that a sporadic link is enabled (valid for a command network) and are not subdivided into packets. Finally it is assumed that all delays (eg. propagation delays, switching delays) are insignificant in comparison to the delay experienced waiting for links to become ‘enabled’.

+ In the section entitled ‘Flooding and Contention’
The SEER System.

A software package, System for the Evaluation of Efficiency of Routings (SEER) was designed and implemented in Pascal, using a clone of the IBM 286 AT. It accepts as input a file describing a network in terms of stations and links, with each link having two numbers associated with it. These numbers define the mean delay of link availability, as well as the standard deviation of this delay.

The system is presented with some number of message sends to be simulated, and randomly assigns sources and destinations for these messages. It then proceeds to simulate both the send of each message and the return of an acknowledgement of its receipt, by each of the three routing strategies. Statistics are kept so that the overall mean delay and delay standard deviation for each strategy can be determined.

Sending a Message

Computation of the time needed to send a message and to receive an acknowledgement is achieved by a simple summing of the delays found for each link on each of the two routes being considered (source to destination and destination to source).

Determining the delay for an attempted send over a particular link is achieved by using the following formula:

\[ d_l = \max \left( md_l + \left( r \cdot \sqrt{3} \cdot sd_l \right), 0 \right) \]

where:
- \( d_l \) is the delay for a send over link \( l \)
- \( md_l \) is the mean delay associated with link \( l \)
- \( sd_l \) is the standard deviation of delay associated with link \( l \)
- \( r \) is a random number in the range -1 to 1.

151
The decision to allow for a range of root three standard deviations on either side of the mean is based on this ensuring that the standard deviation of the generated delays will be the same as the standard deviation associated with the link. (The derivation of this result is given in Appendix A.) The max function is used in order to avoid negative delay time.

An actual send is determined to have occurred if a randomly generated number in the range zero to one is greater than the error rate specified by the user. If the random number is less than or equal to this rate, the attempt fails. The time taken for the send across a link is then taken to be the sum of the delays for each attempted send plus the delay for the actual send. +

Route Determination

Determining shortest-path and minimum-hop routes is simple enough using Dijkstra's algorithm ([6]) for shortest paths. (For the minimum-hop case edge weights of one are given to all links, for shortest-path the mean delay associated with each link serves as its edge weight).

Determining flood routing is a far more complex problem. Clearly determining delays along all possible routes in a network would be impractical for all but the smallest networks. (Consider the thousands of different routes between any given source and destination that would be encountered in a hundred node fully-connected network, for example.) However if the simulation is to be valid it is necessary to consider all routes which have the potential to yield the least delay in sending the message.

This problem was resolved by using what is essentially a branch- and-bound search as described by Lawler and Wood [7], and making use of what Ibaraki [8] refers to as ‘pruning by dominance’.

The algorithm to send a message (or acknowledgement) from source to destination is :

+ Note: the formula for delay given above is recomputed for each attempt, in order to correctly model the sporadic environment.
Set shortest completed path time to infinity
Create an incomplete path from source to destination consisting of just the source node. Set ‘time taken to here’ for this path to be zero
Make this path the first (and only) element in a queue of paths
While the path queue is not empty do
  Set current path to be the front path in the queue
  Remove the current path from the queue
  For each neighbour of the last station on the current path do
    If the neighbour is already on the current path, ignore it (cycle)
    Else
      Determine the delay over the link from last station to the neighbour
      Set NewTime to the current path’s ‘time to here’ plus this delay
      If NewTime is greater than shortest completed path time then do nothing further (not a candidate for quickest flood route)
      Else
        If the neighbour is the destination station then
          Set shortest completed path time to NewTime
        Else
          If the neighbour is on no other path then
            Make a duplicate of current path, and add the neighbour to the path. Set the path’s ‘time to here’ to NewTime, and place it on the queue
          Else
            Compare Newtime to the time taken to ‘time to here’ at the point where the neighbour was reached on the other path.
            If NewTime is greater then do nothing further (duplicate copy being received, so can ignore path), else delete the other path from the queue, and construct and add to the queue the path created by adding the neighbour to current path (in this case the other path would have carried the duplicate rather than the first copy received, and so would not be pursued)
        End (of ‘for each neighbour’ loop)
      End (of ‘if NewTime’)
    End (of ‘for each neighbour’ loop)
Sort queue by ‘time to here’ of paths, with least-cost paths at front of the queue
End (Of ‘while queue not empty’ loop)

This algorithm guarantees finding the shortest route of the flood, as will be explained in the following paragraphs. In addition, it prunes the search space in such a way as to allow computation time to meet reasonable constraints.

By considering all incomplete paths until they become longer (greater delay) than some completed path, or until some station on the path is found to have been reached earlier by another route (in which case the station would have ignored the ‘duplicate copy’ of the message and so

† Greater detail on the degree to which pruning is achieved is given in the section entitled ‘A Note on Speed’ later in this paper.
the path would not have been continued on to the destination station), we ensure that no candidate for the shortest flood path is ignored.

At the same time, by sorting the queue of incomplete paths by time taken (path cost) to date, we ensure that the most promising alternates are explored first. This will tend to allow 'expensive' paths to be safely excluded as candidates early in the search. (These paths will be excluded when some other candidate yields a quicker route to a station on the path, or when the cost of the incomplete path exceeds that of the shortest complete path.)

The structures used to implement the queue and paths are pointer records, as shown below:

![Diagram of Queue Structure]

Figure 1 - Pointer Structure used For Flood Routing
Results

Results are given both for a 6-station network with varying mean delays and delay standard deviations, with delay time much as might be found in Meteor-Burst networks and for a fully-connected 50-station network with all links having identical delay characteristics.

The 'Meteor-model' network is shown below. The numbers on links indicate the mean delay and, in brackets, the standard deviation of delay, in seconds, for each link. Note that the network has been constructed to have reciprocal links, so that only one pair of numbers is given on each connection between each pair of stations, despite there being two links between them.

![Figure 2 - A Test Network](image-url)
Results obtained by the SEER system simulating the transmission of one thousand messages on this network at various error rates are shown in Figures 3 and 4 below (the full numeric results appear in Appendix B):

**Figure 3** - Mean delays of routing strategies at various error rates

**Figure 4** - Standard deviations of routing strategies at various error rates
Considering the lower error rates first, it is obvious that shortest-path outperforms minimum-hop. (In this network not all least mean delay paths are also least intermediary paths, obviously if this were the case the two strategies would perform equally well.) Flooding is significantly better than shortest-path, having a mean delay of only about 75% that of shortest-path, and a standard deviation of delay of about 70% that of shortest path, for the 0.05 error rate. At this low error rate the gains achieved by flooding can probably be attributed to its finding shorter delay paths for particular sends than achieved by using paths with the shortest mean delays.

Now what is most interesting is how robust the flood routing is in the face of higher error rates. At the 0.2 error rate we see that flood now has a mean delay of around 69% of that of shortest-path routing, and a standard deviation of only about 60% of that of shortest-path routing. This relative gain increases as higher error rates are encountered, as can be seen in Figures 5 and 6 below.

![Figure 5 - Flood mean delay as a percentage of minimum-hop mean delay.](image)

![Figure 6 - Flood mean delay as a percentage of shortest-path mean delay.](image)

This evidence that flood routing becomes relatively more attractive the noisier a channel gets can be attributed to the fact that in shortest-path or minimum-hop routing a failed attempt to send will generally result in considerable delay, while in flooding the failure will not have a great effect on delay if the link can be 'bypassed' by an alternate route taking the same time, or only a little more.
This 'bypass potential' also has considerable effect on the standard deviation of delay, as can be seen from the results. This is especially important to situations where 'worst-case behaviour' of a sporadic network is an important concern, such as would be the case in military networks.

The second network tested was a fully connected 50-station network, with each link being assigned a mean delay of 12 seconds and a delay standard deviation of 7 seconds. The simulation was run on 200 messages for a total of 400 messages and acknowledgements being sent.

<table>
<thead>
<tr>
<th>Routing strategy</th>
<th>Mean Delay</th>
<th>Delay Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-Hop</td>
<td>24.95</td>
<td>11.63</td>
</tr>
<tr>
<td>Shortest-Path</td>
<td>25.61</td>
<td>10.63</td>
</tr>
<tr>
<td>Flooding</td>
<td>3.47</td>
<td>1.26</td>
</tr>
</tbody>
</table>

Shortest-path routing and minimum-hop routing are equivalent in this case, with both strategies utilising the direct links between sources and destinations (the minor difference in results being due to the random factors in the computation).

Flood routing gives remarkably better results than the other strategies on this network, with less than 14% of the mean delay and less than 12% of the standard deviation encountered with the other strategies. The mean delay does seem exceptionally low at first glance, however on considering the topology of the network it becomes obvious that such a result should be expected. The range in which generated delays can fall is from zero seconds to a fraction over 24 seconds. This means that we would only expect the delay on the direct link between source and destination to be 'low' (say two seconds or less) about $\frac{1}{12}$ of the time. Two-link paths will have delays in the range zero seconds to 48 seconds, and would thus have delays of two seconds or less about $\frac{1}{24}$ of the time. However, there are 48 two-link paths in this fully-connected 50-node network, so we would expect two of these ($48 \times \frac{1}{24}$) to have such delays. Thus without even considering the other potential paths (3-link, 4-link, etcetera), it is clear that such low delays for flood routing in a highly-connected topology are not only possible but should in fact be expected.
A note on speed

Clearly the most time-consuming section of code in the SEER system is that dealing with the generate-and-test involved in the flooding simulation. The performance of the algorithm used shall be briefly discussed.

The algorithm has a best case scenario occurring when there is a direct link between source and sink, and this link has the least delay amongst all the links emanating from the source. Here the algorithm will detect the optimal flood path after inspecting only the links connecting the source to its neighbours, which gives a gain over exhaustive search equal to $E_s/E_n$, where $E_s$ is the out-degree of the source, and $E_n$ is the number of edges on all source-sink paths.

The worst-case scenario would occur in the unlikely situation where all paths between source and sink are node-disjoint (no common intermediaries would prevent any pruning by dominance) and all path delays from source to the station immediately preceding the sink are greater than the shortest complete path from source to sink. This latter condition means that no partial path could be discarded until its entire delay was computed. In this worst-case scenario the search algorithm would give no gain whatsoever over exhaustive search.

Determining an average case would be a particularly difficult exercise, if indeed it is possible to do so. Clearly the topology of the network and the pattern of delays on links between source and sink (do ‘bad’ paths become distinctly bad early?) would have to be considered in any such determination. However the algorithm presented here has constrained the problem well for the networks tested - for 400 floods on the fully-connected 50-station network discussed the simulation ran to completion in under an hour, while the simulations on the smaller network were handled in seconds. This on a 286 PC/AT clone.
Flooding and Contention

Queing contention arises when a station has a number of messages to send, and the order in which they are sent will affect the delay associated with their transmission. Such contention can arise under any of the routing strategies - one needs only consider the hub of a star topology network to see this. However it is clear that the flood mechanism will be most prone to create such contention, and so we will consider only this case in determining at what point such contention would degrade network performance.

An important point as regards sporadic networks is that queing contention is only significant if the time period that the link is enabled between some transmitter T and receiver R is too short to allow all messages T has for R to be sent. (Given the earlier assumption that the delay between link availability is our only significant delay, it follows that there is no significant delay between a message transmitted immediately the link becomes viable, and one transmitted immediately before it is no longer viable.) In the worst case scenario of flooding, T would have a copy of every message in the network waiting to be sent to R when the link becomes enabled, and so contention would arise if the link was not enabled for a sufficient time to allow all these messages to be sent.

From this, we can see that a number of factors affect whether contention arises in a sporadic network or not, (and how severely it becomes a problem). Firstly, the amount of data in the network as a whole is a concern. The network can avoid contention if it is loaded to any level up to any point where the throughput possible on a link is greater than or equal to the amount of data being transmitted in the network. As loading becomes heavier than this so contention could arise, and the greater the loading the higher the risk, and the more severe the effect, of queing contention.

Conversely, the greater the throughput potential of enabled links, (the amount of time they’re enabled for, times the data-rate they can support), the higher the loading that can be achieved before contention arises.
Finally, of course, the topology of the network and the pattern of link availability will determine how severely loaded individual stations become. (ie. how bad our worst-case for a particular network can get.)

As an example, in the case of Meteor-Burst Communications, a usable trail would be roughly 250 ms, and could support a data rate of about 32 kbps. This means that an enabled link would be capable of sending around 8 kilobits of data. For application networks with an average message length of 40 bytes or so this would be about 25 messages. Thus the network could be at risk of queing contention if more than 25 messages were ‘in transit’ at any given time.

Determining the probability of such contention actually occurring when the network is loaded beyond this ‘safe’ level, and to what degree it would affect the efficiency of routing strategies, is a subject that needs further work.

Conclusion

The design of the SEER system, in particular in terms of its handling of the problem of sporadic link availability and the branch-and-bound algorithm used to simulate flooding, has allowed for effective modelling and fast simulation of routing in a sporadic network environment.

Flood routing has clearly yielded significantly better results for sporadic networks than the other two alternates considered, particularly in terms of both lower mean delay in the network as well as far lower standard deviation of delay. It becomes even more attractive in situations where transmission error rates are high.

Shortest-path routing will have significant benefits over minimum-hop routing in some networks, but will obviously be equivalent in networks where the minimum-hop route and the shortest-path route are the same.
There are of course other factors to be considered in contrasting the routing strategies - overheads such as the construction of routing tables, queing contention, problems such as buffer overflow and so on. However it does seem that the particular nature of sporadic networks will still make flood routing the most effective routing strategy for these networks.

Further work is required in two areas. The effect of queing contention on delays experienced, as has been discussed, requires more study. In addition, a study of the effects of network topologies on routing efficiency would be of great interest. It is clear from the results that a high degree of connectivity will greatly increase the relative efficiency of flood routing over the two other strategies considered. A generalised formulation of the decrease in delay as a function of the increase in connectivity would be of great use to network designers, as would be the investigation of routing performance over a wider range of basic network topologies. (Stars and linked-stars, hierarchical networks, and so on.)

Acknowledgements

The author wishes to express his gratitude to SALBU (Pty) Ltd and the FRD for making funding available without which this work would not have been possible.
References

[1] Larsen, J.D., Melville, S.W. and Mawrey, R.S.M. ‘Adaptive Data Rate Capacity of Meteor-Burst Communications’ in Conference Record of the 1990 IEEE Conference on Military Communications (MILCOM90) at Monterey, California, Vol 2, pp 40.1.1 - 40.1.5., (1990)


Appendix A - Statistical Derivation

Having a mean $\mu$ and a standard deviation $\sigma$ of delay associated with each link, the simulator required that random numbers be generated over an interval in such a way as to ensure that the numbers generated had the same mean and standard deviation as were associated with the link. The derivation of this interval is described here.

Now a random distribution over an interval will have the same mean and standard deviation as a uniform distribution, given that we have an infinite number of samples. Clearly then the centre of the interval must be the mean associated with the link, and we seek to determine $k$ such that an infinite number of random numbers generated in the range $(\mu - k \sigma, \mu + k \sigma)$ will have a mean equal to $\mu$ and a standard deviation of $\sigma$. As we have a uniform distribution, we may consider the equivalent range of $(0, 2k\sigma)$ and a corresponding mean of $k\sigma$.

We now use a discrete uniform distribution with $n+1$ points to derive $k$, and then determine the result as $n$ tends to infinity. We shall work with variance rather than standard deviation for ease of computation.

The $n + 1$ points occurring uniformly across the range 0 to $2k\sigma$ are

$$0, 1, \frac{2k\sigma}{n}, 2, \frac{2k\sigma}{n}, 3, \frac{2k\sigma}{n}, ..., n, \frac{2k\sigma}{n}$$

Their mean, $k\sigma$, can be written as $\frac{n}{2} \cdot \frac{2k\sigma}{n}$, giving us the following formula for their variance:

$$\frac{1}{n+1} \sum_{i=0}^{n} \left( \frac{2k\sigma}{n} \left( i - \frac{n}{2} \right) \right)^2$$

164
\[
\begin{align*}
  &= \frac{1}{n+1} \sum_{i=0}^{n} \left( \frac{4k^2 \sigma^2}{n^2} \left( i^2 - ni + \frac{n^2}{4} \right) \right) \\
  &= \frac{1}{n+1} \left( \frac{4k^2 \sigma^2}{n^2} \frac{n^2}{4} + \sum_{i=1}^{n} \left( \frac{4k^2 \sigma^2}{n^2} \left( i^2 - ni + \frac{n^2}{4} \right) \right) \right) \\
  &= \frac{1}{n+1} \left( k^2 \sigma^2 + \frac{4k^2 \sigma^2}{n^2} \left[ \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2} + \frac{n^3}{4} \right] \right) \\
  &= \frac{1}{n+1} \left( k^2 \sigma^2 + \frac{4k^2 \sigma^2}{n^2} \left[ \frac{4n^3 + 6n^2 + 2n}{12} - \frac{6n^3 + 6n^2}{12} + \frac{3n^3}{12} \right] \right) \\
  &= \frac{1}{n+1} \left( k^2 \sigma^2 + \frac{4k^2 \sigma^2}{n^2} \left[ \frac{n^3 + 2n}{12} \right] \right) \\
  &= \frac{1}{n+1} \left[ k^2 \sigma^2 + \frac{k^2 \sigma^2(n^2 + 2)}{3n} \right] \\
  &= k^2 \sigma^2 \left( \frac{1}{n+1} \left[ 1 + \frac{n^2 + 2}{3n} \right] \right)
\end{align*}
\]

Let this equation equal \( \sigma^2 \)

then \( k^2 \left( \frac{1}{n+1} \left[ 1 + \frac{n^2 + 2}{3n} \right] \right) = 1 \)

\( \Rightarrow k^2 \left( \frac{1}{n+1} \left[ \frac{n^2 + 3n + 2}{3n} \right] \right) = 1 \)

\( \Rightarrow k^2 \left( \frac{1}{n+1} \left[ \frac{(n+1)(n+2)}{3n} \right] \right) = 1 \)

\( \Rightarrow k^2 \frac{(n+2)}{3n} = 1 \)

\( \Rightarrow k^2 = \frac{3n}{(n+2)} = 3 \left( \frac{n}{n+2} \right) \)

Now, \( \lim_{n \to \infty} \frac{n}{n+2} = 1 \)

so \( \lim_{n \to \infty} k^2 = 3 \)

\( \Rightarrow k = \sqrt{3} \)
## Appendix B - Numerical Results of Routing Simulation

<table>
<thead>
<tr>
<th>Error rate = 0.05</th>
<th>Routing strategy</th>
<th>Mean Delay</th>
<th>Delay Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-Hop</td>
<td>75.82</td>
<td>56.74</td>
<td></td>
</tr>
<tr>
<td>Shortest-Path</td>
<td>63.98</td>
<td>43.11</td>
<td></td>
</tr>
<tr>
<td>Flooding</td>
<td>50.30</td>
<td>34.56</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error rate = 0.1</th>
<th>Routing strategy</th>
<th>Mean Delay</th>
<th>Delay Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-Hop</td>
<td>76.11</td>
<td>57.12</td>
<td></td>
</tr>
<tr>
<td>Shortest-Path</td>
<td>64.49</td>
<td>49.38</td>
<td></td>
</tr>
<tr>
<td>Flooding</td>
<td>51.50</td>
<td>34.70</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error rate = 0.2</th>
<th>Routing strategy</th>
<th>Mean Delay</th>
<th>Delay Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-Hop</td>
<td>99.83</td>
<td>69.53</td>
<td></td>
</tr>
<tr>
<td>Shortest-Path</td>
<td>81.79</td>
<td>62.80</td>
<td></td>
</tr>
<tr>
<td>Flooding</td>
<td>59.22</td>
<td>34.85</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error rate = 0.5</th>
<th>Routing strategy</th>
<th>Mean Delay</th>
<th>Delay Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-Hop</td>
<td>134.05</td>
<td>121.94</td>
<td></td>
</tr>
<tr>
<td>Shortest-Path</td>
<td>125.78</td>
<td>112.09</td>
<td></td>
</tr>
<tr>
<td>Flooding</td>
<td>74.58</td>
<td>54.87</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error rate = 0.8</th>
<th>Routing strategy</th>
<th>Mean Delay</th>
<th>Delay Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum-Hop</td>
<td>344.23</td>
<td>323.26</td>
<td></td>
</tr>
<tr>
<td>Shortest-Path</td>
<td>256.63</td>
<td>219.20</td>
<td></td>
</tr>
<tr>
<td>Flooding</td>
<td>144.72</td>
<td>124.99</td>
<td></td>
</tr>
</tbody>
</table>
The Design of a Speech Synthesis System for Afrikaans

M.J. Wagener

Department of Computer Science, University of Port Elizabeth,
PO Box 1600, Port Elizabeth, 6000

ABSTRACT

The design of a speech synthesis system for the generation of speech from text is presented. The system accepts any Afrikaans sentence and then derives segmental and suprasegmental information through various linguistic processes. The resultant information is used to generate control parameters for a formant synthesiser.

Relevant features of the most well-known experimental speech synthesis systems are given, followed by a specification of the design objectives of the system. The system design is explained in terms of data modelling and data flow analysis. Further explanation is provided for the implementation of the data structure, the data structure manager and a typical linguistic process.

1 Introduction

The synthesis of speech by a computer can be done in different ways, ranging from the simple reproduction of previously recorded speech to the synthesis of speech from text. Methods based on prerecorded speech are easy to implement and provide high quality sound but are restrictive in terms of what speech can be reproduced. Speech synthesis from text on the other hand, requires complex implementation algorithms with a lower speech quality but is completely flexible in its application. The system presented here falls in the latter category. It takes unrestricted text and uses linguistic knowledge to synthesise the speech. The process consists of two main phases;

- the translation of text to a phonetic representation and
- the generation of a speech signal from the phonetic representation.

2 Overview of speech synthesis systems

The earliest speech synthesis systems concentrated on the production of a speech signal from a given phonetic representation. Various types of synthesizers were developed which can be used to generate phonetic sounds. These synthesizers fall into two broad classes namely articulatory and formant synthesizers. The operation of articulatory synthesizers is based on the physical human articulation movements whereas formant synthesizers attempt to reproduce the acoustic signal associated with speech. This system and most other experimental speech synthesis systems make use of a formant synthesizer.

During the 1970's research was directed towards the automatic translation of text to a phonetic representation. In order to derive sufficient information from text, linguistic processes like syllabification and stress assignment must be applied. Carlson and Granström [Car76] developed the first system which separates the linguistic knowledge from the logic of the system. They developed a special programming language that allows the specification of linguistic knowledge in terms of rules. Other systems like the Klattalk system developed by D.H. Klatt [Kla82], also followed this approach but went further in modularizing the system in terms of separate linguistic processes. This can also be seen in the SRS system developed by S.R. Hertz [Her82]. The SRS system uses three different rule sets to perform specific parts of the translation process.

A major shortcoming of these systems is that they all operate on a linear data structure which is semantically overloaded. This has a direct and restrictive effect on the expressive power of the linguistic rules. A further shortcoming is that
the systems are geared for the synthesis of a specific language and fall short as general synthesis environments.

These two shortcomings were addressed in the development of the Delta system [Her85]. The Delta system is hailed as the ultimate synthesis system. It uses a delta (an hierarchically interconnected structure) as the central data structure and a powerful programming language to manipulate it. The user is required to do all data typing and structuring and manipulate the delta structure by following pointers and coding loops and procedures. The expertise required by a user is certainly on par with that of a third generation programmer. The Delta system thus addresses relevant issues concerned with the engineering of a speech synthesis system, but in trying to be a general speech synthesis environment, it has turned out to be another specialised third generation language.

3 Design objectives

The above discussion of existing speech synthesis systems gives an overview of their historic development and identifies the major design issues that were addressed in these systems. With this overview in mind, the following objectives were identified as important in the development of a speech synthesis system for Afrikaans. The objectives are given with reference to the existing systems.

- To provide a framework whereby the linguistic theory for the generation of Afrikaans speech can be practically implemented and tested. The emphasis is on designing a speech synthesis environment specifically for Afrikaans. It differs in this aspect from the Delta system which caters for different languages. Although the system is aimed at a restricted application domain, it must still be general and flexible enough to be used as an experimental environment for synthesising Afrikaans speech.

- To provide a system in which the linguistic knowledge is clearly separated from the logic of the system. This objective was already achieved by Carlson and Granström and also in the later systems, but the Delta system has gone backwards by combining the specification of linguistic knowledge and the logic of traversing the data structure.

  - To provide a friendly and familiar interface to the linguistic rule writer. The system is aimed at linguists with none or very little programming experience and must thus provide an interface that suits the expertise and needs of this type of user. The Delta system requires knowledge of programming concepts like data typing, data structuring, loops and modularisation, which makes it unsuitable for use by pure linguists.

  - To employ an internal data structure that closely models the real world and provides manipulative power. The use of a linear data structure in the earlier systems like the Klattalk and SRS systems, was one of the major drawbacks of these systems. This was addressed in the Delta system by introducing the so-called delta. The criticism against the Delta system is that it does not provide any abstract data model based on the delta, but leaves it to the rule writer to do the semantic data modelling. This level of expertise is not expected from rule writers using this system.

  - To provide a functional decomposition of the system that is organised into levels of abstraction and semantics with clear interfaces between these levels. This objective is to a certain extent already a solution to some of the problems raised earlier and fundamental to the design of modern software systems.

Throughout this paper, special emphasis is placed on explaining how these objectives influenced the design of the synthesis system.

4 Designing the data structure

It was specified that the internal data structure used in the system should represent the user's perspective of the data. A top-down approach is followed whereby firstly the user's view of the data is determined by an abstract data modelling
technique. The abstract data model is then implemented in a physical data structure.

The sentence is taken as the synthesis unit because larger units like paragraphs, do not contain more information that is derivable by the system. The structural components of a sentence which are of interest for this study are words, syllables and sound segments. The sound segments further consists of parameter frames which contain the parameter values for the formant synthesizer. An abstract data model that represents all the required data elements and the relationships between them, is constructed by using entity-relationship modelling [How83]. The entity-relationship model is shown graphically in an entity-relationship diagram in figure 1.

All the structural components of a sentence namely words, syllables, sound segments and parameter frames are all represented in the model by the SEGMENT entity. This is possible because they are all described by the same attributes. Consequences of this generalisation are firstly that the model is very simple and therefore easy to manipulate and secondly that the model is flexible in that other types of components, e.g. clauses and morphemes, could also be accommodated in the SEGMENT entity. Each occurrence of the SEGMENT entity can be associated with certain features. All the features of interest to the system are combined into one entity, called the FEATURE entity. This again puts no restrictions on the different types of features that can be accommodated.

The relationship CONSISTS-OF represents the internal structure of the data e.g. one word consists of many syllables. The HAS relationship shows that a segment can have many features but also that it has one value for that associated feature.

5 Data flow analysis

Figure 2 shows a data flow context diagram of the system. The rule writer provides linguistic rules which are used to generate control parameters for a synthesiser from a sentence entered by the user.

The flow of data inside the synthesis system is shown in Figure 3. All processes operate on the data structure through the data structure manager. The data structure is initialized with information from the input text, expanded by various participating linguistic processes and finally used to generate control parameters for the synthesizer.

The processes are modularized according to their linguistic function in the system. Processes 2, 3 and 4 work together to generate the segmental information for the speech signal. Segmental information applies to individual sound segments. The function and existence of these processes are justified by the theory of natural generative phonology [Com87]. The transformation of a phonological sound segment to a phonetic sound segment as depicted in the design, stands central to this theory. Processes 5, 6 and 7 generate suprasegmental or prosodic information. Prosodic information has a global effect on speech and applies to larger segments like sentences and words.

The different processes in the design are now explained in more detail.

5.1 Text normalization

This process serves as a filter of the text to convert the input to a standard form, e.g. expand abbreviations and numerals like “i.p.v.” to “in plaas van” and “1” to “een”. The standard form accepted by the system consists of only words and punctuation symbols combined into sentences. Any other special characters which are not handled in the normalization rules will be filtered out. The user can thus customize this module to suit his own needs.

5.2 Generating sound segments

This process takes as input the orthographic characters from the data structure and generates corresponding phonological representations. It also assigns initial feature information to the phonological sound segments. The process closely corresponds with what is known as letter-to-sound rules in other systems [Wag87], except that the output is an abstract phonological sound segment on which further phonological rules must be applied before a phonetic representation is acquired.
5.3 Syllabification

This process groups the sound segments into syllables and assigns feature information to each syllable. A syllable typically consists of vowel which is optionally surrounded by consonants. Since phonological rules can change the syllable boundaries, the syllabification rules must be applied again after the phonological rules.

5.4 Application of phonological rules

It is the task of this process to convert the underlying phonological representations to phonetic representations. This is achieved by different types of phonological rules that insert, delete and replace sound segments and also expand the sound segment feature information. It uses as input the existing sound segments and syllables as well as the existing feature information of these entities.

5.5 Partial syntax analysis

The current aim of the partial syntax analysis of a sentence is to determine the syntactic role of each word in the sentence. Since a natural language parser is beyond the scope of this study, this is achieved by maintaining a dictionary of words with their parts of speech. The dictionary is searched for each word in the sentence and the part of speech is added to the feature information of the word.

5.6 Sentence stress assignment

The syntactic role of words and punctuation symbols are used to stress certain words in the sentence and insert different pauses in appropriate places in the sentence.

5.7 Word stress assignment

This process uses the syllabic information in the data structure to apply a stress pattern within a word. Syllables within a word are given contrasting degrees of stress. Word stress and sentence stress information together is used in the next process to generate fundamental frequency, duration and amplitude parameter values.

5.8 Parameter generation and interpolation

The function of this process is to use the segmental and suprasegmental information generated by the other linguistic processes and determine control parameters for the synthesizer. Parameter values of neighbouring sound segments are interpolated to provide smooth transitions. The parameters are targeted at a software formant synthesizer [Kla80] that is used to construct a digital representation of the speech signal.

6 Implementing the data structure

The physical implementation of the abstract data model has a file for the SEGMENT entity and a file representing the HAS relationship between SEGMENT and FEATURE. Each segment is represented by a record in the SEGMENT file and each feature of a specific segment is stored as one record in the SF file. (Note that these files can also be seen as a sequence of records stored in memory.) The relationships between these records (eg. one segment has many features) are implemented by keeping pointer fields in the records. The record formats of the two files are:

SEGMENT(information, parent_segment, first_segment, last_segment, left_segment, right_segment, first_sf)
SF(name, value, parent_segment, right_sf)

In the SEGMENT file the number field is represented by the physical record number and is therefore excluded from the record description. The CONSISTS-OF relationship is implemented by three types of pointers; a bidirectional child link, a child-parent link and a parent-to-first-and-last-child link. The bidirectional child link connects segments which are all part of the same larger segment together and allows traversal of
these segments in both directions. The child-parent link connects each segment to the segment in which it is contained, and the other two pointers connect a segment with the first and last segments of its list of subsegments. The relationship between segments and features is implemented by a pointer chain connecting firstly a segment to its first feature (the first.sf field in SEGMENT) and then each feature with the next (the right.sf field in SF). Each feature is also connected to its parent segment by the parent.segment field in SF. An example of part of the physical data structure for the sentence “Ek eet lekker” is given in figure 4. A feature list is shown for the word “ek”.

7 Data structure manager

The data structure manager plays a very important role as interface between the data structure and the rest of the system. It provides an abstract view of the physical data structures and hides any physical changes that might occur in the data structures, from the rest of the system.

The abstract view provided by the data structure manager can be described as a hierarchical structure of segments in which each segment can have a number of features. The interface is formally defined in terms of the data structure operations provided by the data structure manager. The abstract data structure and defined operations closely follows the approach followed in traditional network database management systems as proposed by the Data Base Task Group [Dat86]. It hides all the intricacies of pointer management from other components in the system and provides complete and powerful operations on the abstract data structure.

Each segment and feature has a pointer or address associated with it. The interface allows access to these pointers which can then be used to navigate through the data structure by means of the operations. The example shown below serves as an illustration of typical operations defined in the data structure manager. In this example the feature “byw 1” is added to the last word in the sentence in figure 4. Assume access to the address of the sentence segment is obtained prior to this code and stored in PARENT. CHILD is another pointer variable and FOUND a boolean variable.

FOUND := find_first_seg(CHILD, PARENT)
while FOUND do
  FOUND := find_right_seg(CHILD, CHILD)
  insert_first_feat(CHILD, ’byw’, 1)

The data structure manager provides the other components in the system with data independence. If any changes in the physical storage of the data are made, it would not affect the other components. These changes are absorbed by the interface. It should also be noted that the interface does not enforce the semantics of speech synthesis but provides a framework that is ideal for such an implementation. The actual structuring of a sentence into words, syllables, sound segments and parameter frames is semantics enforced by other higher level components. Thus the data structure manager in turn is protected from changes in these semantics.

8 A typical linguistic process

The internal decomposition of each of the linguistic processes shown in Figure 3 is the same. It consists of a rule compiler and an inference engine. The interdependence of these components is shown in figure 5. The user enters linguistic rules into an ASCII file using a text editor. The rules are analysed for syntactic errors and then compiled into object code (which is Pascal). The inference engine uses the output from the rule compiler and manipulates the data structure via the data structure manager.

Further attention is firstly given to the format of the linguistic rules which will give an indication of the interface provided to the rule writer, and secondly to the operation and reasons for using an inference engine.

8.1 Rule format

The format of the linguistic rules is consistent with the format of phonological rules used in the linguistic literature [Com87].
A rule has a target level in the hierarchical data structure associated with it. The target level depends on the linguistic process in which the rule is used. The phonological rules for instance will operate on the sound segments. The general function of a rule is to identify a specific segment in the data structure and then to modify the information and features of the segment. A segment is identified by specifying the information content and/or feature information of the segment. Depending on the specific linguistic process, the changes specified by the rule is applied to the same or next lower level of the hierarchy. A change can be an insertion, deletion or update of the current segment information. A segment can also be identified by specifying a context in which it must appear. The context can include one or more segments to the left and/or right of the target segment. Matching any segment in the rule can also be preceded and/or followed by one or more statements. These statements are primarily used to assign the result of Pascal expressions to predefined string, real and integer variables.

8.2 Inference engine
Each of the different linguistic processes in the system uses an inference engine to apply linguistic rules to the data structure. The algorithms used in these inference engines only differ in terms of the specific level of the data structure that they operate on. The main task of the inference engine is to traverse through the appropriate level of the data structure, choose a rule to apply and provide the procedures which are called in the object code.

The inherent nature of the inference engines enforces a specific semantic model on the data structure. The inference engines use the data structure manager to build a data representation that reflects the linguistic structure of the input data. By operating on a specific level of the data structure, an inference engine adds a specific semantic interpretation to that level. The parser for instance, takes its input from the first level and generates segments for the second level of the data structure. This implies that the first level represents the sentence and the second level the different words from the sentence. If the synthesis model is changed, the changes will be absorbed by the inference engines and it will not affect the data structure manager.

Another important reason for using an inference engine is to separate the linguistic knowledge, which is contained in the rule sets, from the logic of the system.

9 Conclusions
In conclusion the system design is discussed in terms of the original design objectives.

The system caters for all the relevant linguistic processes involved in the translation of text to speech and is therefore suitable for the practical evaluation of these processes. It will thus be possible to determine if processes like syllabification aids in the synthesis of intelligible speech.

A clear distinction is made in the system between the specification and application of linguistic knowledge. All the linguistic information is represented in the rule sets and modifying these rules does not affect the logic of applying them to the data structure.

It is further maintained that the system provides a friendly and familiar user interface because

- it uses a rule format that is consistent with the rule format used in the linguistic theory.
- for each rule set the rule writer operates on a data model that is appropriate to the specific linguistic process and equivalent to the data model perceived in the linguistic theory.

The data structure closely models the real world and can thus be easily manipulated. A major advantage of the implementation of the data structure is the two levels of semantics associated with it, namely the physical level provided by the data structure manager and the logical level provided by the inference engines. This provides the necessary abstraction in terms of the user interface but at the same time provides a flexible system which can be adapted for different synthesis strategies and the synthesis of other languages.

A final comment regarding the design is that it lends itself ideally to a parallel implementation. The different linguistic processes can be executed in parallel with communication amongst
them handled through the central data structure manager. This will be pursued in a later version of the system.

References


[Dat86] Date, C.J. 1986. 'An Introduction to Database Systems.' Volume 1, Addison-Wesley.


[How33] Howe, D.R. 1933. 'Data Analysis for Data Base Design.' Edward Arnold.


Figure 1: ER Diagram

Figure 2: DFD Context diagram
Figure 3: DFD Overview diagram
Figure 4: Example of physical data structure

Figure 5: DFD Level 1 diagram
EXPERT SYSTEMS FOR MANAGEMENT CONTROL: A MULTIEXPERT ARCHITECTURE

VEVEK RAM
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF NATAL
PIETERMARITZBURG

ABSTRACT

The use of Expert Systems technology in management decision making domains is increasing rapidly as business environments worldwide grow more turbulent and as the cost of development tools decrease. Research effort in this field however, is concentrated largely on confined areas such as market analysis, financial diagnosis and production scheduling. The development of an Expert System to support a wider management area presents problems of both size and complexity since such a system would require a large monolithic knowledge base which would exhibit the associated problems of maintainability, consistency and reduction in inference speed.

This paper describes a blackboard based Multiexpert architecture that is capable of integrating the problem solving capabilities of a range of confined expert systems in order to provide problem solving support for a wide area such as management control at the strategic level. The system consists of several dedicated expert modules in the area of marketing, finance, production and so on as well as a control module that handles problem decomposition, task allocation and dynamic scheduling. A prototype version of such a system has been successfully implemented in Prolog.

INTRODUCTION

Many organizations engage in Long Term or Strategic Planning in order to match their internal capabilities with the opportunities and threats that exist in their operating environments. Such a match is characterised by the commitment of the organization's resources to achieve a desired objective and is often referred to as the organization's Strategic Posture. Management Control is the process whereby the organization continually re-assesses the appropriateness of the match and re-aligns its Strategic Posture to accommodate changes in the environment. The rate at which the organization is able to respond to changes in the environment is known as the Strategic Response Rate.

The intensification of global competition has emphasised the importance of a rapid Strategic Response Rate as only those organizations that are able to seize environmental opportunities early can compete effectively. Unfortunately, shifts in Strategic Posture involve the whole organization and the effects of it have to be assessed in various areas before changes can be implemented. This generates a lag in the response. For most organizations, posture shifts involve a reorganization of marketing, financial, production, research and human resources plans. Since these areas are separate in most organizations, the response lag can be attributed to the actual delay in assessment in each area and also the delay that can arise due to the communication and co-ordination between these departments. Computer-based systems in the form of Decision Support Systems and Expert Systems have to a large extent provided assistance.
in reducing the problem in the individual areas. Descriptions of such systems can be found in King and Rodrigues (1977), King and Dutta (1980), Klein and Newman (1980), Bouwman (1983), Smith et al (1985), Cooper (1986), Chandrasekaran and Ramesh (1987, 1988), Goul (1987), Lee and Lee (1987) and Biswas (1988). Although the problem of communication and co-ordination can be solved by developing a single system that is representative of the collective activities of the various organizational areas, such a system would require a large and complex knowledge base. Large knowledge bases exhibit problems of maintainability and consistency (Prerai et al 1990). Also, there is a considerable reduction in inference speed and efficiency as the size of a knowledge base increases. The other alternative is to build a system that can integrate the functions of various individual systems by enabling them to co-operate to solve a common problem while at the same time retaining their individual status. This approach is used extensively in the area of Distributed Artificial Intelligence and many useful techniques have been developed as a result. Specifically, the blackboard architecture (Erman et al (1980), Nil (1986) and Hayes-Roth, (1988)) and the centralised multi-agent framework (Cammarata, (1983)) is most readily applicable. The blackboard architecture is based on a shared global data structure called the blackboard. The blackboard is divided into levels of varying abstraction depending on the application. Independent knowledge sources may read from and write to one or more levels of the blackboard. Multiagent frameworks use a single agent or knowledge source or a group of knowledge sources to form a coherent plan for solving a multiagent problem. Dependencies and potential conflicts among the agents are identified in advance. In centralised multiagent frameworks, one agent acts as the controller and coordinator for the whole network of agents. A combination of the centralised multiagent framework and the blackboard architecture facilitates the integration of discrete knowledge sources and enables the power of their collective knowledge to be used as a single large knowledge base without the associated problems. The remainder of this paper describes the architecture and operation of such a distributed system for use in the management control area. Construction details for some of the more important aspects of the system is also included.

ARCHITECTURE

The distributed management control system consists of a control module, a scanning module and several functional modules as shown in Figure 1. A brief description of the knowledge sources or modules and a discussion of their major roles in the distributed network follows.

The Control Module.
The control module acts as the strategic management expert and also as the manager of the network. As the strategic management expert, it controls the direction and format of the network problem solving process. It contains knowledge about the strategic management process and it also contains meta-knowledge, which is knowledge about how the rest of the system's knowledge is distributed throughout the network. This meta-knowledge allows the control module to decide that interest rates concern the financial expert, product cost concerns the production expert and so on. As the network controller, the control module controls the execution of individual modules as well as the management of the status of the blackboard.
The Scanning Module.
The scanning sub-system acts as the machine interface between the network and the organization. The scanning module monitors a set of strategic factors and reports all variances to the control module via the blackboard. It performs a simple but nevertheless important role in the network. The module is non-intelligent in that it reports all variances. The control module decides on the severity of an occurred variance.

The Functional Knowledge modules.
There are currently four functional modules in the prototype network. Each module contains conceptual knowledge of the domain area in general and also of specific policies of the organization in that area. The domain areas are Marketing, Finance, Production and Organizational. Each module has sufficient domain and control knowledge in order to function as a stand alone expert or knowledge-based system. Additional modules for other functional areas such as Research and Development or Distribution can be added when required.

OPERATIONAL OVERVIEW

Each knowledge source is responsible for maintaining a set of strategic variables in its own domain. Variables are categorised as either internal or external depending on whether the entity that a variable relates to is changed from within or outside of the organisation's boundary. As long as the values of these variables remain within predefined limits, there is a balance between the organisational ability, environmental pressure and a chosen strategy. Values for the internal variables are held in the organisational database which is constantly updated through the organisation's information system. External variables are updated through manual input on a regular basis. All variables are monitored by the scanning subsystem. When the value of a variable changes, the scanning subsystem communicates this change to the control module. The control module decides on the degree of severity of the variance (and others which may occur simultaneously), assigns priorities and then decides on which
modules need to be called in order to resolve the problem. It then posts a request with parameters describing the nature of the variance on the blackboard and activates the appropriate expert module or knowledge source. The individual knowledge source assesses the impact of the change in relation to the present strategic posture and communicates the result back to the control module via the blackboard. If the result concerns other knowledge sources, these are then activated by the control module. The process continues until a final result is obtained that is consistent with all the experts individual results. If two or more experts put forward recommendations that are conflicting, the control module can resolve the conflict by choosing the recommendation with the highest utility value or by modifying and reposting variables on the blackboard so that the individual experts reassess their respective recommendations and in so doing resolve the conflict themselves after a number of cycles.

COMMUNICATION AND CONTROL

The System Blackboard.
The blackboard does not exist as a physical entity in the system but rather as a communication mechanism through which the knowledge sources communicate both their requests and their findings. All individual knowledge source activity is initiated from the blackboard and all conclusions or results from knowledge sources are directed to the blackboard. In this application, the blackboard is in the control of the strategy formulation and control knowledge source. The system blackboard is divided into three main areas as shown in Figure 2.

![Figure 2. The System Blackboard](image)

These three areas are used for static knowledge, dynamic knowledge and control knowledge respectively. Static knowledge is the domain specific knowledge that is relevant to the problem and normally remains relatively stable during the solution process. In the system blackboard, the static knowledge area holds the collection of organizational data that is scanned by the scanning module. Dynamic knowledge is knowledge that is generated during the execution of the system. It consists typically of new facts, hypotheses and suggestions that are made by the knowledge sources. In the
system blackboard, the dynamic knowledge is divided into the generic strategy area and the functional strategy area. The generic strategy area holds heuristic suggestions that are made by the control module and which are used by the functional knowledge sources to restrict their search spaces. The functional strategy area contains all the functional strategies, which are the solutions to a strategic problem, that are generated by the individual sources. Control knowledge is knowledge about the current state of the network itself and also of the status of the problem solving. In the system blackboard, the control knowledge is made up of a set of requests which form a dynamic queue. The requests are either from the control module to a functional module or vice-versa. The control module extracts from this request list a single request which it then converts into a call to an individual module. The responsibility of keeping the blackboard "clean", that is, erasing old or unwanted entries or archiving previous entries rests with the control module. This is an essential activity since the blackboard tends to become cluttered after a reasonable amount of network activity and this can lead to a degradation of the network efficiency.

Scheduling and Control
Network control can be achieved by selecting an individual knowledge source and calling on it to execute inside a problem solving cycle, or it can be achieved by placing knowledge on the blackboard that will cause a knowledge source to execute on its own. The support system uses the strategy formulation knowledge source as the network controller and therefore makes use the former method. The network as a whole makes use of three control mechanisms: Goal-Driven control, which is the control exerted on the network to attain a network-wide or global system goal; Request-Driven control, which is the control exerted on the network by inter knowledge-source requests, and Event-Driven control, which is the control exerted on the network due to the occurrence of certain events. The goal-driven function of control is the classical strategic management function of monitoring and controlling of strategic and functional plans. The goal of the network is to ensure that the implemented strategy adheres to certain performance limits that were used in the formulation of the strategy. Variances that exist obviously affect the strategic posture as a whole and must be accommodated at the strategic level. Variances are translated into network action by the event-driven control mechanism. A variance is regarded by the scanning subsystem as a strategic event that triggers the network into a resolution process. The event-driven control function is to alert the control module into initiating the network. Individual knowledge sources would then attempt to reduce the variance or the effects of it and failing this, to reevaluate the strategy. The request-driven mechanism allows the control module as well as other modules to alter the direction of the problem solving process in a dynamic way by posting requests on the blackboard. These requests may be for further information or for initiating the activation of other modules. There may be many such requests on the blackboard at any one time and in a serial network, that is a network in which the knowledge sources cannot be executed in parallel, these requests need to be serviced in some sort of order. This is accomplished by establishing a schedule of ready-to-be-called modules. This schedule must be flexible enough to be dynamically modified since the execution of one module may cause others that were ready to execute to be no longer required or, the execution of one module may cause others that were not executable, ready to execute. Also, there may be more than one consecutive request for the same module, and the schedule must allow a single activation call with all the
requests consolidated rather than allow more than one consecutive call to the same module.

IMPLEMENTATION

The prototype version of the distributed management control system has been implemented in Prolog on a microcomputer. As the exact constructional details are beyond the scope of this paper, only the more unusual aspects of communication, task decomposition and dynamic scheduling are described. Also, as each individual knowledge module is a conventional production rule and frame based Expert System these will not be examined. Further detail can be found in Ram (1990a, 1990b).

Communication.
The mechanism used to control and facilitate the flow of information in the network is the blackboard. The blackboard is a communication and storage mechanism which is accessible by all the modules and which is divided into different levels. Prolog has an internal database which can be modified during execution and can also be stored and retrieved. This forms a convenient implementation of the blackboard. The different levels can be distinguished by using a separate predicate for each level. At the static level which holds organizational data for example, the predicate has the form:

Factval(FactorName,value)

An example of such a clause is Factval("InterestRate","LOW").

The control level of the blackboard is used to hold control information for the scheduling mechanism. This information is in the form of request clauses that are inserted into the blackboard at the control level by those functional modules that require assistance or additional information. Requests are held in the following format

Request(CallMod,DestMod,Factor,Action,Ref,Status)

CallMod identifies the module issuing the request, DestMod is the module to which the request is directed, Factor and Action are as before, Ref is a request reference number and Status indicates the status of the request. A status value of "U" denotes unresolved and a status value of "OK" or "NotOK" denotes a resolved request. When the control module assembles requests into a queue, it examines the status value in each request and ignores requests that have already been resolved. An example of a request is

Request("MARK","PRICE","DEC",1,"U").

This is a request from the marketing module requesting that the control module investigate the possibility of a reduction in product price. Requests in the individual modules are typically invoked by rules which test for the existence of required data. The DestMod slot is left blank since the individual modules do not have knowledge of each others expertise. The control module, through its decomposition procedure, decides on the module to which it can best delegate the resolution of the request and fills the DestMod slot before the delegated module is called. If the request can only be resolved by more than one module, the control module issues as many requests as the decomposition procedure generates.

Problem Decomposition
When an individual functional module encounters a subproblem during its problem solving activity that is outside its domain of expertise, it would issue a request to the control module for assistance. It is the function of the control module to redirect these requests to the appropriate modules. A major problem for the control module in the
execution of this function is "knowing" which module to call for a given request. A simple and effective way to overcome this problem is to maintain a list that links all the relevant organizational data items with the modules responsible for them. Such a list represents Meta-level knowledge since it represents knowledge about the use of the distributed expertise in the most efficient way. When a request that can be resolved by a single module is received, the control module need only scan the list in order to identify the module best suited to resolving the request. A problem arises when a request is received that cannot be resolved by one module alone. Such a request has to be decomposed into subrequests that can be resolved by individual modules. This decomposition process can be implemented by organizing the decomposition relationships into a taxonomy of meta-knowledge frames as follows.

MFrame(Problem, PRef, DecompList, Dmodule)

where:
MFrame is a label distinguishing the Meta-Knowledge frames from other frames in the program;
Problem is the label identifying the problem that this instance of the frame is representing;
PRef acts as a reference number for the problem represented by this frame and is used to establish priorities in the problem solving process;
DecompList is a list of all the subproblems that Problem can be decomposed into;
Dmodule is the Domain module responsible for solving Problem and is only present in a frame if the DecompList contains a single element, or if it contains more than one element, then all these elements are the responsibility of the same domain module.

Consider as an example, that the scanning module has picked up a drop in the market share. One option for strategic realignment is to restore the situation by stimulating primary market demand which expands the total market or by stimulating selective demand which increases market share within the existing market. A marketing action plan of reducing product price or increasing advertising can achieve both these. Since product price is outside the domain of the marketing module, it will request the control module to investigate the feasibility of price reduction. The control module has to refer this request to the appropriate module or modules and makes use of the meta-knowledge frame taxonomy search to decide which module or modules are appropriate. The search begins by finding a frame which has price as the label in the problem slot.

MFrame("Price",1,"["Cost","Margin"], )
The PRef slot is arbitrarily set to 1 and the domain slot is empty since DecompList contains more than one element. This frame represents the decomposition of the price problem into the two subproblems of cost and margin. The search then continues by finding a frame for each of the elements in the DecompList. These are found as

MFrame("Cost",1,"["ProdCost","Ohead"], )
MFrame("Margin",1,"["Margin"],FIN)
The first frame further decomposes the cost problem into the two subproblems of production cost and production overhead. The second frame asserts that margin cannot be decomposed further and that it is the responsibility of the FIN or Financial module. The control module continues the search by finding frames with Prodcost and Ohead as labels in the problem slot. This produces the following frames

MFrame("Prodcost",1,"["ProdCost"],PROD)
MFrame("OHead",1,["OHead"],FIN)

Since both these frames contain only one element in their respective DecomLists, the search terminates and the control module posts a request to the PROD or production module to investigate the reduction in product cost. The production module contains rules that relate product cost to raw material and labour costs and so is able to function independently in solving this subproblem. The control module also posts a request to the FIN or financial module to investigate the possibility of a reduction in profit margin and production overhead. Both the financial and the production modules communicate the results of their investigations to the control module. Both the requests derived from the decomposition have the same PRef number as the original request and the scheduling mechanism uses this number to keep them in the same logical group.

Control of Dynamic Scheduling

Dynamic Scheduling is accomplished by establishing and managing a queue of Ready-to-be-called modules. The queue is represented by a prolog list and is constructed by examining all the requests held in the control level of the blackboard. The scheduling procedure terminates when the queue is empty which occurs when there are no unresolved requests on the blackboard. Once a queue has been constructed, the control module calls the functional module represented by the first entry in the queue. When the call terminates, that is, when the functional module has completed its task, the control module then reconstructs the queue and the process is repeated. Reconstructing the queue each time a functional module call is terminated, ensures that the scheduling mechanism makes use of the most current problem solving knowledge available. This is necessary since at any stage, a called module may issue a request and suspend its problem solving activity until the request is resolved. The module chosen for investigating this new request must be inserted at the head of the queue and called. On its termination, the original module which is waiting for the response is called and continues its task.

CONCLUSION

A description of a distributed knowledge-based management control system has been presented. A prototype version which has been implemented in Prolog, has generated very favourable results in an important area. It is hoped that the success on this limited scale will encourage further research in other areas. It is considered that the most important aspect of this work is the illustration that complex areas requiring knowledge-based support can be structured into relatively self-contained knowledge modules which can then be integrated into a system which, while addressing the original problem, is easier to build, debug and maintain.

REFERENCES


Cammarata, S. et al, (1983), Strategies of Cooperation in Distributed Problem Solving, Proceedings of the IJCAI.

Chandrasekar, G. and Ramesh, R., (1987), Microcomputer Based Multiple Criteria Decision Support System for Strategic Planning, Information and Management, 12, 163-172.


Ram, V., (1990b), A Distributed Knowledge-Based Support System for Strategic

Integrating Similarity-Based and Explanation-Based Learning

G. Deon Oosthuizen    Chrisna Avenant

Department of Computer Science
University of Pretoria
0083 Hillcrest, South Africa
E-mail: rkw0075.upvm2@f4.n494.z5.fidonet.org

Abstract

Recently, there have been various attempts to combine the strengths of similarity-based learning (SBL) and explanation-based learning (EBL) in a single learning system.

We describe a graph-based learning method called Graph Induction, which is based on the graphical representation of a formal lattice and supports both supervised and unsupervised learning. The method integrates SBL with a weak form of EBL in such a way that the two mechanisms become totally blended. The result is a unified algorithm with both SBL and EBL involved in each step. The domain theory is generated and/or extended as SBL proceeds and employed immediately, through EBL, to guard further learning and thus control the size of the lattice which otherwise has the potential for increasing exponentially.

Keywords: Artificial Intelligence, Machine Learning

1. INTRODUCTION

Two major Machine Learning paradigms are Empirical and Analytical learning. Empirical learning induces rules from a number of examples, referred to as the training set. Since many of the methods used are based on the similarities between samples, they have become known as Similarity-Based Learning (SBL) methods. Analytical learning employs domain theory to induce a class description from one sample only. Since this was viewed by many authors as a process of learning by explaining why a sample belongs to a given class, it has become known as Explanation-Based Learning (EBL).
Recently various attempts have been made to combine the strengths of SBL and EBL [Segre 1989]. We describe a representation model which allows the two paradigms to be integrated, complementing each other during each learning step. Langley points to the fact that the two paradigms are more similar than the literature suggests [Langley 1989]. The current paper harbours the same sentiment.

The representation model described forms the basis of an incremental inductive learning method, called Graph Induction, implemented in a system called GRAND (GRaph iNDUction) [Oosthuizen 1987]. Conceptual clustering forms the basis of generalization in GRAND. Thus, it is primarily a concept formation method [Oosthuizen & Avenant 1991], extending the ideas embodied in the UNIMEM [Lebowitz 1986a] and COBWEB [Fisher 1987] systems. However, GRAND also supports concept learning as a special case of conceptual clustering by regarding the class of a sample - i.e. the concept to be learnt - as just another feature. This places GRAND among systems like AQ15 [Michalski et al 1986] and ID5 [Utgoff 1988]. GRAND is also closely related to the CHARADE system [Ganascia 1987].

In this paper, we focus on the interplay between similarity-based and explanation-based learning taking place in GRAND. In this regard, GRAND has much in common with work done by others [Lebowitz 1986b], [Pazzani 1988] and [Vilain et al 1990].

We first describe the representation model and then explain how SBL and EBL are supported and integrated. We conclude by comparing GRAND with related systems.

2. GRAPH INDUCTION

Graph Induction derives its name from the fact that it is based on the explicit graphical representation of a formal lattice. We now explain how the lattice is constructed and how it supports inductive learning.

2.1 Construction of the lattice

A lattice is an acyclic directed graph in which every pair of nodes has a least common superior (the join) and a greatest common subordinate (the meet) which are necessarily unique. A lattice is constructed by first creating a node for each value of each attribute to be recorded in the system. These nodes can be considered as the upper layer of a network to be expanded below them. Training samples are read in the form of arrays of features - normally attribute-value pairs. For the first sample (array) read, a single new node is created below the initial layer of features and connected to each of the attribute values (in the initial layer) constituting the sample (see fig. 1a). Thus, the sample is represented by a node linked to each of the attribute values composing the sample.
The same is done for the second sample read, with the difference that if the second sample has any features in common with the first sample, the graph is transformed such that the lattice properties - mentioned above - are preserved. I.e. each subset of attribute nodes would have a unique meet (if it does have a meet) and each group of samples would have a unique join (if it does have a join). In the process intermediate nodes are created between the attribute-nodes at the top and the sample-nodes at the bottom. The intermediate nodes are referred to as concepts. The nodes above a given node are said to be spanned by the node and the ones below are covered by it. For each set of samples there exists a unique corresponding lattice\(^1\). (Since the focus in this paper is on the application of lattices to learning, the ideas are discussed with reference to informal examples only. A comprehensive description of the transformation algorithm can be found in [Oosthuizen 1991] and another application is described in [Oosthuizen 1990].)

Let us consider the following example. A number of elephants (see Table 1) are classified on the basis of 4 features each. The attributes are ear size, colour, temperament and love for candy. Fig. 1 shows the concept nodes (indicated by "*-nodes) created during transformation. Notice that this is only a partial graph. Some arcs have been deliberately omitted for the sake of simplicity.

![Diagram of concept nodes](image)

**Fig. 1.**

\(^1\)To construct a proper lattice, the attribute nodes have to be connected to a single common superior, and the sample nodes to a common subordinate. We omit them since they have no role to play in the conceptual modelling exercise.
<table>
<thead>
<tr>
<th>Sample no.</th>
<th>Ear_size</th>
<th>Colour</th>
<th>Temperament</th>
<th>Loves_Candy</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>small</td>
<td>pink</td>
<td>fierce</td>
<td>yes</td>
<td>African</td>
</tr>
<tr>
<td>S2</td>
<td>big</td>
<td>pink</td>
<td>friendly</td>
<td>no</td>
<td>Indian</td>
</tr>
<tr>
<td>S3</td>
<td>big</td>
<td>grey</td>
<td>fierce</td>
<td>yes</td>
<td>African</td>
</tr>
<tr>
<td>S4</td>
<td>small</td>
<td>black</td>
<td>fierce</td>
<td>no</td>
<td>Indian</td>
</tr>
<tr>
<td>S5</td>
<td>big</td>
<td>black</td>
<td>fierce</td>
<td>yes</td>
<td>Indian</td>
</tr>
<tr>
<td>S6</td>
<td>big</td>
<td>pink</td>
<td>fierce</td>
<td>yes</td>
<td>African</td>
</tr>
<tr>
<td>S7</td>
<td>big</td>
<td>black</td>
<td>friendly</td>
<td>no</td>
<td>Indian</td>
</tr>
<tr>
<td>S8</td>
<td>small</td>
<td>pink</td>
<td>friendly</td>
<td>no</td>
<td>Indian</td>
</tr>
<tr>
<td>S9</td>
<td>big</td>
<td>pink</td>
<td>friendly</td>
<td>yes</td>
<td>Indian</td>
</tr>
</tbody>
</table>

Table 1.

2.2 Similarity-Based Learning

We now explain how the lattice supports generalization and clustering.

Each intermediate node in the lattice is associated with a cluster of features at the top and a cluster of samples at the bottom. Thus, each intermediate node denotes a concept characterized by the attribute values spanned by the node (its intension). Below each intermediate node is a cluster of samples exemplifying the concept (its extension).

The lattice captures all similarities between samples in a series of tangled hierarchies. Stepping bottom-up, each concept covers a larger set of samples. Thus the hierarchies form sequences of concepts of increasing generality (upward). If a specified set of samples has a mutual join, then this join is unique and constitutes a least generalization (maximally specific concept) of the set.

2.2.1 Concept Descriptions

Each node can be regarded as an identity (label) for a class - the set of samples below it. The characteristic description of the class is the set of features spanned by the node. In that sense, the concept 'nodes represent induced class descriptions or 'rules'. Although the features spanned by a node can be read as a list, the specific tree structure involved reflects dependencies between features. Consequently, each node denotes a rule [Oosthuizen & McGregor 1988]: if the (unique) meet of a number of features spans any additional features, apart from the given features themselves, then such features are inferred. For example, in fig. 1 the meet of PINK, FIERCE and CANDY_LOVER\(^2\) is *10, and *10 spans AFRICAN in addition to the named features, therefore:

PINK and FIERCE and CANDY_LOVER  ->  AFRICAN

Thus, the lattice itself can be used as basis for inference, i.e. as knowledge base. This has

\(^2\)For the sake of brevity we write PINK, instead of COLOUR=PINK, etc.

190
the additional effect that training and prediction are integrated, i.e. the system learns incrementally while it operates and there is no distinction between training and test instances. Each instance confirms certain patterns and contradicts other patterns associated with nodes stored in the lattice.

A large number of generalizations are kept in the lattice and evaluated in parallel, treated as embryonic concepts with the potential to be confirmed, or to be contradicted and eliminated (see next section). The lattice structure changes continuously and forms the basis of SBL.

As we mentioned above, GRAND provides maximally specific generalizations. When a positive instance of the concept is seen but the current concept definition would classify it as a negative instance, the concept definition is redefined to be the intersection of the current concept definition and the instance [Sarrett and Pazzani 1989]. This corresponds to the Wholist strategy [Bruner, Goodnow & Austin 1956], also called the One-Sided Algorithm for Pure Conjunctive Concepts [Hausler 1987]. As in Bruner's work, the initial hypothesis is a conjunction of all features in the first positive example. Sarret & Pazzani, however, initialize the hypothesis to be the conjunction of all features in the example description language.

As already intimated, the intermediate nodes represent concepts with associated features. The coincidence of features associated with a concept gives rise to dependencies between the features. The coincidences and dependencies between features in turn give rise to predictability, which forms the basis of inference, and in the case of 'strong' concepts (see next section), to inference rules. Because of this ambiguity in the nature of the intermediate nodes, we will sometimes refer to them as concepts and sometimes as rules, depending on their role in the particular context.

### 2.2.2 Confidence Factors

Incidental similarities between samples give rise to the creation of a multitude of intermediate nodes representing insignificant clusters. To keep the size of the lattice within reasonable limits, a pruning strategy is applied [Oosthuizen 1991] whereby insignificant nodes (nodes denoting patterns of features which are not confirmed) are removed on a regular basis. However, if the number of features per sample is large (15+) pruning alone becomes inadequate to curb the lattice. The lattice size is much better contained if the SBL is directed by EBL (see next section). Pruning is based on a confidence factor (CF) associated with every node. The CF of a node is an integer value equal to the number of samples read containing the pattern of attribute values spanned by the node, i.e. the number of sample nodes below it; we refer to it as the strength of the node.

By specifying a certain threshold value, nodes can be divided into a set denoting established rules and a set representing weaker dependencies between features which
may or may not become rules depending on whether they are confirmed or not. (Nodes which are not confirmed, are removed eventually.) This is similar to Lebowitz' idea of *freezing* a feature in conceptual clustering [Lebowitz 1986a]. Lebowitz distinguishes between the confirmation of a feature and of a generalization. The CF of a feature is used to determine whether the feature forms part of a generalization description. The strength of a generalization is used as a measure of interest (or "relevance") to decide when EBL must be applied to a generalization. In other words, the number of instances a generalization represents is an indication of its relevance or "interestingness" (after Lebowitz). Both these notions of confirmation of features and of generalizations are incorporated in the CFs in GRAND. E.g. *15 represents a generalization identified by BLACK, INDIAN covering instances S4, S5 and S7. Although *5 represents the generalization FIERCE, BLACK, INDIAN the CF of *5 really indicates the number of times BLUE occurred in conjunction with BLACK and INDIAN. If this happens often enough, *5 would be frozen, thus identifying FIERCE, BLACK, INDIAN as a useful or permanent generalization with BLACK, INDIAN as a more general generalization. Otherwise, *5 would be deleted, leaving the latter generalization only. The same would happen in UNIMEM by changing the CF of the feature FIERCE.

2.3 Explanation-Based Learning

In Section 2.2 we explained how SBL is supported by the lattice. We now consider EBL, which is manifested as follows. Confirmed rules are regarded as forming part of the domain theory which has evolved. Most of these rules express dependencies between features. In other words, they are not part of the description of the concept to be learnt. E.g. in fig. 1, the node *9 denotes the rule BIG and FIERCE -> CANDY_LOVER which does not form part of the descriptions for the classes INDIAN or AFRICAN. Since there are many nodes in the lattice denoting such rules, it implies that the lattice contains a rich theory of the domain. (Notice that because of the existence of *24 and *6, the following inferences are inhibited: FIERCE and CANDY_LOVER -> BIG BIG and CANDY_LOVER -> FIERCE)

The intermediate nodes can also be viewed in a slightly different way. An intermediate node which spans a pattern which occurred many times, can be viewed as an eminent concept, identified by the system, but which is not listed among the recorded features. The detection of such concepts has been studied under the heading of feature construction [Matheus 1990]. If questioned, a domain expert (or oracle [Muggleton, 1987]) might be able to associate a lexical term with it. In graph terms: such a node may be viewed as being linked to an additional feature, not present among the given features. E.g. let us
again consider *9 in fig. 1. Considering the fact that these elephants are fierce, big-ear, candy loving elephants, *9 could easily be linked to a new feature SPOILT_ELEPHANT. *9 would thus denote the rule

**BIG and FIERCE and CANDY_LOVER** -&gt; **SPOILT_ELEPHANT**.

This interpretation of the graph differs from the one above in the sense that here *9 suggests the existence of an autonomous concept, defined by the features spanned by the node *9. In other words, the node *9 denotes a concept which constitutes the right-hand side of a (domain theory) rule which has as its left-hand side the features spanned by the node itself. This is the reason why the intermediate nodes were referred to as concepts in the first place (see section 2.1). Thus, each (confirmed) concept node can be visualized as having an arc to a corresponding lexical label (feature) at the top.

If a subset of features of a sample is spanned by a particular concept, then it means that the sample satisfies the rule describing the concept. And if this concept in turn implies the target concept (i.e. the concept being learnt - INDIAN in this case), then these rules explain why the sample is an instance of the target concept. Thus, when a new sample is read, it is first determined whether it contains one or more subpatterns of features (which may overlap) which are spanned by eminent concepts in the lattice\(^3\). This constitutes the first step of explanation-based learning, i.e. explaining why the sample is an instance of the concept being learnt.

E.g. let us say the following sample is added to the lattice in fig. 1

S10: **BIG, GREY, FRIENDLY, NON-CANDY_LOVER, INDIAN**

Let us say *13 denotes the concept GOOD_ELEPHANT, described by the features\(^4\)

**BIG, FRIENDLY, NON-CANDY_LOVER**.

The fact that *13 is below (i.e. a subset of) INDIAN, implies that all GOOD_ELEPHANTS are INDIAN elephants. Consequently, the fact that S10 satisfies the conditions for GOOD_ELEPHANT explains its being an instance of INDIAN. Although, in one sense, *13 denotes one rule incorporating all the features above it, in another sense *13 and the nodes above it - *7, *26, *3, *21 and *18 - all denote interrelated rules. If a sample satisfies some of these rules, these rules are chained together to explain why the sample is an instance of the target concept.

The second step involves using the domain rules to identify important and irrelevant features of the sample with the aim of generalizing it. If there are any of the sample’s features that are not spanned by eminent concepts, they are likely to be irrelevant - especially if we are talking of lattices containing several hundreds of concepts. Thus the sample is not connected to such features, which effectively removes a part of the sample description and thereby makes it more general (see Section 2.5.2). In other words, we

\(^3\)This happens implicitly as part of the transformation algorithm.

\(^4\)INDIAN is omitted from this list since in the example it is the concept being learnt.
accomplished explanation-based generalization.

Since GREY, in the above example, is not spanned by eminent concepts, it is dropped, and the sample is generalized by connecting it to *13 only.

Thus, while SBL is in progress, the domain theory evolves - in the form of implicit rules in the lattice. As soon as these are confirmed adequately, they begin to play a role in EBL. The benefit of this type of learning is that knowledge necessary to perform EBL can be acquired by the learning system and the learner gets better at learning [Pazzani 1988].

2.4 External Rules

In the previous section we considered a domain theory that was developed internally from scratch. Domain theory can be added to the lattice in the form of rules supplied by an external source. Rules are treated just like samples - ignoring the separation between antecedent and consequent parts but associating extraordinary high confidence factors with the rule nodes. Such rules are then automatically incorporated in the learning process, just as the internally generated ones are. Thus:

- samples read are generalized by means of both SBL and EBL [Pazzani 1988], but simultaneously
- the domain theory is generalized or specialized through SBL, and
- new domain theory is developed through SBL.

Pazzani [1988] refers to specially selected examples inserted with the specific aim of establishing the domain theory as "foundational examples".

2.5 Discussion

2.5.1 Effect of EBL on the lattice size

SBL has the effect of generating intermediate nodes and EBL has the effect of removing them (referred to as operationalization) [Wogulis & Langley 1989]. Although the above weak form of EBL does not remove nodes from the graph, it does prevent nodes from being created and thus enables us to contain the lattice. E.g. let us say a concept *1 represents an adequately confirmed pattern <A,B,C,D> (see fig. 2a) and a pattern <A,B,D,F> is read as part of a sample S_n, where C and F are different values of the same attribute. If normal transformation takes place, a new node *4 is created to represent the new pattern of features (see fig. 2b). However, if EBL takes place, *1 is identified as the meet of <A,B,D>. Since *1 denotes a strong rule, we accept that A and B and D -> C.

Thus, C is inferred and F ignored and no new nodes are created (see fig. 2a).
2.5.2 Comparing Similarity-based and Explanation-based Generalization

In SBL, there are several ways to generalize a concept [Michalski 1983]. Because of the graphical nature of Graph Induction, the disjunction of concepts is represented by joins. E.g. the arcs from *27 and *9 to *24 in fig. 1 implies that *24 denotes the concept description

BIG and CANDY_LOVER and (INDIAN or FIERCE)  
(the set of elephants denoted by *24 are BIG and FIERCE, and either INDIAN or FIERCE or both) which is the disjunction of the concepts denoted by *27 and *9. Consequently, all generalization involving range extension or merging reduces to "climbing the generalization tree" [Michalski 1983]. Within this restricted representation framework, the similarity between SBL and EBL becomes more apparent.

SBL involves generalization using a rule of the kind

A -\rightarrow B

meaning: generalize a concept description by replacing a concept A by a more general concept (descriptor) B. In the lattice, B would be a parent node of A.

EBL involves generalization using a rule of the kind

A and B and C -\rightarrow D

meaning generalize a concept description

by replacing the expression by a more general concept (descriptor) D. The fact that the concept representing <A,B,C> is an explicit node in the lattice, and the node denoting concept D is a parent node of it, just as B is a parent node of A, illustrates the similarity between the basic generalization mechanisms of EBL and SBL.
3. RELATED WORK

Although various related systems have been referenced already, we now focus on some of them specifically.

3.1 UNIMEM

As we mentioned before, GRAND extends the ideas implemented in UNIMEM [Lebowitz 1986a] - the main difference being that UNIMEM uses a hierarchy where GRAND uses a lattice as framework for knowledge representation. Lebowitz [Lebowitz 1986b] explains how external rules can be used to apply EBL to concepts, in particular to establish causal relationships between features. To apply EBL methods to UNIMEM, rules have to be supplied that capture the initial understanding of the domain. In UNIMEM this is done with implications that capture hypothesized low-level causal connections among features. Rules (external from the hierarchy and hand coded) can be used to explain the presence of one feature from the presence of another feature. With the initial rules UNIMEM can engage in EBL with the purpose of analyzing a generalization done by SBL. The relevance of a generalization determine whether EBL will be applied to it [Lebowitz 1986b]).

In GRAND, such low-level rules can be added to the concept-base (lattice) itself and the explanations can be derived in a similar way. However, the kind of low-level dependencies added by Lebowitz as domain theory, are normally the first kind of dependencies which quickly evolve in the lattice. In other words, most of them would be there already. Like OCCAM [Pazzani 1988], (but unlike UNIMEM) GRAND can thus be regarded as a "closed-loop" learning system where the same memory is used for SBL and EBL.

Secondly, UNIMEM stores features as 'flat' lists associated with generalizations. In GRAND, each feature is involved in a complex graph structure reflecting its dependence on other features or groups of features, as well as features with which it jointly determines other features. In other words, whereas UNIMEM has to apply an algorithm to derive explanations, in GRAND they can be read from the graph. Since the lattice stores all the low level dependencies between predictive and predictable features, the explanation can be derived by a straightforward interpretation of the graph structure.

In summary, UNIMEM requires a separate EBL procedure to identify causal relationships from features contained in generalizations, while in GRAND the EBL takes place implicitly and the causal relationships are there to be read. UNIMEM uses the confidence factor of a generalization as a measure of its interestingness and uses this to determine which generalizations to explain. In GRAND every generalization can be explained.
3.2 OCCAM

OCCAM [Pazzani 1988] also integrates SBL and EBL in one system. The main differences between GRAND and OCCAM are
- Pazzani's emphasis on knowledge-intensive strategies,
- the knowledge representation used, and
- the differentiation between Theory-Driven Learning and Explanation-Based Learning - a result of his use of Conceptual Dependency Theory.

OCCAM starts with an initial hierarchy of schemata which represents the conceptual dependency actions, goal and states; as OCCAM learns, the hierarchy is extended by creating specializations of the existing schemata. Thus, OCCAM incrementally forms a concept hierarchy that explains and organizes previous experiences [Pazzani 1988].

Pazzani puts much emphasis on the fact that the more knowledge-intensive strategies have stronger justifications and should receive priority in learning. (Empirical techniques justify the inclusion of a feature in a schema on the simple basis that it has appeared in previous events.) Since EBL is more knowledge-based whereas SBL is more statistics-based, there is merit in the argument that EBL should be given priority. However, although the argument about the superiority of knowledge intensive learning is a valid one, it has to be remembered that, ultimately, knowledge has to come from somewhere - this role of extracting knowledge from data is fulfilled by SBL.

3.3 Analytical and Similarity-based Classification

Vilain et al [1990] present an exposition on the role of classification (both analytical and similarity-based) in Knowledge Representation and Machine Learning. They contend that pre-inserted domain knowledge is essential "in providing an inductive bias to the learning procedure, thereby shortening the required training phase, and reducing the brittleness of induced generalizations". Since KL-ONE, the knowledge representation scheme used by them, is also lattice based, it is no surprise that there is a high degree of correspondence with GRAND. Vilain et al divide features characterizing generalizations ('generalization frames') into definitions, and norms which are interpreted as defaults. There is a direct correspondence between their definitions and the features spanned by a node in GRAND. Similarly, their norms correspond to the features spanned by nodes immediately below a given node in GRAND. E.g. the definition of *15 (see fig. 1) would be

GREY and NEGATIVE

and its norms would be the nodes spanned by *5 and *12 not spanned by *15, namely BLUE and TALL.

In GRAND the norms are either inferred probabilistically or treated as proper defaults as in other knowledge representation paradigms. Depending on which interpretation is adopted, additional features spanned by S4, S5 and S7 (which are not immediate children of *15) might also be regarded as norms.
Since concepts in GRAND are (implicitly) not subject to the canonical form imposed on the analytical language of Vilain et al, GRAND is more expressive. GRAND also utilizes a different classification method. But for the rest there is a remarkable correspondence in functionality between the two approaches.

3.4 Incremental Version Space Merging

Hirsch [1989] uses the version space paradigm [Mitchell 1985] to combine empirical and analytical learning. The central idea is to apply explanation-based generalization to training data, and then to do empirical learning on the generalized data. Thus, rather than updating the version space (doing empirical learning) with single instances, each instance has the effect of multiple instances. Since the version space is also incorporated in the lattice, Incremental Version Space Merging and GRAND exhibits similar behaviour along a spectrum from knowledge-poor to knowledge-rich learning.

4. CONCLUSION

"A central activity of science is the search for unifying principles that account for apparently diverse phenomena within a single framework" [Langley 1989]. The lattice model goes some way toward achieving that goal. It has been shown elsewhere [Oosthuizen & Avenant 1991] that the lattice supports supervised and unsupervised learning, rule-based and case-based, incremental and non-incremental learning. In this paper we showed that both SBL and a weak form of EBL are supported.

We explained that Graph Induction does not involve separate SBL and EBL procedures. SBL and EBL are integrated as follows: intermediate SBL results - tentative domain theory - are used as part of an EBL process which is in turn used to guide SBL. Furthermore, SBL and EBL continuously re-validate and extends the already constructed theory. Instead of separate SBL and EBL phases the two processes operate in parallel and in complete synergism. The result is that each kind of learning gains the maximal effect from the intermediate results of the other. Also EBL help control the size of the lattice, expanded as a result of SBL.

Acknowledgements

We want to thank the referees for their comments. We are indebted to Prof. Roelf van den Heever for providing us with the opportunity to conduct the current research, and to Jesus Christ the Origin of Knowledge.
5. REFERENCES


Efficient Evaluation of Regular Path Programs

Peter T. Wood
Department of Computer Science
University of Cape Town
Rondebosch 7700, South Africa

Abstract

The next generation of query languages for database systems should have the ability to express recursive queries, the efficient evaluation of which will be crucial to the success of these systems. One such query language which has been the subject of much research is Datalog. We define a class of Datalog programs, namely, the regular path programs, which can always be evaluated efficiently, in particular, when constants are present in a query. Efficient evaluation is ensured by reducing the number of arguments appearing in each predicate defined in the program. The class of regular path programs is incomparable to previous classes to which the technique of argument reduction has been applied.

1 Introduction

There is little doubt that the next generation of database systems should support query languages that are more powerful than those usually associated with relational databases. The language Datalog is one of those that has received much attention in this regard [Ullm88]. Datalog has a Prolog-like syntax, but, unlike Prolog, is evaluated bottom-up rather than top-down. The reasons for this are to ensure termination of queries and to exploit the efficiency of relational algebra for operations on sets of tuples [Ullm89].

Example 1.1 Consider the following example taken from [Naug87]. Assume that the database comprises two relations: \( \text{likes}(X,Y) \), which states that person \( X \) likes product \( Y \), and \( \text{knows}(X,Y) \), which states that person \( X \) knows person \( Y \). Suppose that people buy the products that they like, or those bought by someone they know. Then the following program \( P \) defines the relation \( \text{buys}(X,Y) \) of people \( X \) and the products \( Y \) they buy.

\[
\begin{align*}
\text{buys}(X,Y) & : = \text{likes}(X,Y). \\
\text{buys}(X,Y) & : = \text{knows}(X,Z), \text{buys}(Z,Y).
\end{align*}
\]

Note that this program is recursive, in that \( \text{buys} \) is computed in terms of itself. This is a capability not found in traditional query languages.

If we are interested in the entire \( \text{buys} \) relation, then we phrase the query \(? - \text{buys}(X,Y)\) against the program. In order to evaluate this query bottom-up, the system would undertake an iterative process known as semi-naive evaluation, in which successive approximations to the complete \( \text{buys} \) relation are computed. These approximations represent larger and larger subsets of the \( \text{buys} \) relation, with the final iteration producing the complete answer. The first approximation is the \( \text{likes} \) relation itself. Successive approximations are found by joining previous approximations of \( \text{buys} \) with \( \text{knows} \) (based on the common attribute \( Z \)) until no new tuples are found. □
One disadvantage of bottom-up methods is their inability in certain cases to "focus" the computation. For instance, if in the above example we wanted to know what John buys, rather than what everyone buys, then we would specify the query $- \text{buys}(\text{john}, Y)$ instead of $- \text{buys}(X, Y)$. However, a simple-minded bottom-up evaluation would still compute the entire buys relation before selecting out those tuples whose first component is John. What is needed is a method analogous to pushing select operations into relational algebra expressions [Ullm88], but one which works for recursive programs.

Top-down methods are usually better in this respect. In our example, a top-down method would first find which products John likes, then whom John knows along with the products they like, and so on. One way to achieve such behaviour for bottom-up evaluation is to transform a given program so that its bottom-up evaluation mimics a top-down evaluation. This is the goal of a method known as Magic Sets [BMSU86,BR87].

Example 1.2 Given the above program $P$ along with query $- \text{buys}(\text{John}, Y)$, Magic Sets produces the following:

\[
\begin{align*}
\text{magic}(\text{john}). \\
\text{magic}(Z) & : - \text{magic}(X), \text{knows}(X, Z). \\
\text{buys}(X, Y) & : - \text{magic}(X), \text{likes}(X, Y). \\
\text{buys}(X, Y) & : - \text{magic}(X), \text{knows}(X, Z), \text{buys}(Z, Y). \\
? - \text{buys}(\text{john}, Y).
\end{align*}
\]

The purpose of the first two rules is to compute all possible bindings of the first argument of buys that would be used in a top-down evaluation. This "magic" predicate is then substituted into each of the original rules in order to restrict the bindings in a bottom-up computation. □

However, the Magic Sets transformation in the above example is still not the most efficient possible. It is not hard to see that the recursive rule for buys is now redundant, since the transitive closure of the knows relation starting with John is computed by the second magic rule, from which all the products bought by John are found by the nonrecursive rule for buys. We can therefore transform the program to the following:

\[
\begin{align*}
\text{magic}(\text{john}). \\
\text{magic}(Z) & : - \text{magic}(X), \text{knows}(X, Z). \\
\text{buys}(\text{john}, Y) & : - \text{magic}(X), \text{likes}(X, Y). \\
? - \text{buys}(\text{john}, Y).
\end{align*}
\]

Notice that the number of arguments in the recursive predicate in the program (initially buys, now magic) has been reduced from two to one. This technique, known as argument reduction or factoring, has been applied previously to various classes of Datalog programs [NRSU89a,NRSU89b]. In this paper, we apply the technique to a different class of programs, namely, the regular path programs.

In the next section, we begin by defining the class of regular path programs. Section 3 is devoted to the application of argument reduction to regular programs when constants are specified in queries. Our claim that this leads to efficient evaluation of such programs is explored in Section 4, where it is shown that the transformed programs are often considerably less time consuming to evaluate than the original programs. Conclusions and topics for future research are discussed in Section 5.

202
2 Regular Path Programs

In the previous section, we saw one example of a regular path program; we now define the class of such programs exactly. The term regular path program is derived from the fact that there is a correspondence between these programs and the problem of finding paths which satisfy a given regular expression in a labelled, directed graph. This connection is explored in more detail in [Wood90], although it should be noted that the regular path programs defined below differ from the programs defined in that paper.

Essentially, we want to generalise programs such as that given in the previous section in two ways. Firstly, we would like to have more complicated rule structures as demonstrated by the following example.

Example 2.1 Assume we have a semantic network in which the nodes represent instances, classes or properties, while edges from instances to classes are labelled with isa (classification), edges between classes are labelled with ako (generalisation), and edges from either instances or classes to properties are labelled with can, has or is. The following program finds the relationship i.inherit between instances and the properties they inherit.

\[
i.inherit(X,Y) \rightarrow isa(X,Z), c.inherit(Z,Y).
c.inherit(X,Y) \rightarrow ako(X,Z), c.inherit(Z,Y).
c.inherit(X,Y) \rightarrow can(X,Y).
c.inherit(X,Y) \rightarrow has(X,Y).
c.inherit(X,Y) \rightarrow is(X,Y).
\]

Note that the above program displays a natural correspondence to the notion of a regular grammar. □

A second way in which we would like to generalise programs is by adding arguments to predicates in order to pass more information among the rules. The following example demonstrates this ability.

Example 2.2 Suppose we have an application involving a network with values labelling the edges connecting nodes. Assume that this information is stored as an edge relation e(X,Y,Z), which states that there is an edge from X to Y labelled Z. If we want to find all nodes that are connected by routes on which at most two alternating values are used, the following program will suffice.

\[
s(X,Y,U,V) \rightarrow e(X,Z,U), t(Z,Y,U,V).
t(X,Y,U,V) \rightarrow e(X,Z,V), s(Z,Y,U,V).
s(X,Y,U,V) \rightarrow e(X,Z,U), e(Z,Y,V).
\]

Note that the program still has a regular structure and contains mutually recursive predicates s and t. The variables U and V are used to hold the pairs of edge labels, while the alternation of values is captured by the mutual recursion. □

Before we proceed, we need to introduce some standard terminology. Given a rule such as

\[
\text{buys}(X,Y) \rightarrow \text{knows}(X,Z), \text{buys}(Z,Y).
\]
the predicate to the left of : — is called the head of the rule, while those to the right constitute the body of the rule. Variables in a program are denoted by strings with an initial upper case letter (e.g. \( X \)), while constants are numeric values or strings having an initial lower case letter (e.g. \( john \)). As is common with Datalog, we assume that all rules are safe, that is, every variable appearing in the head of a rule also appears somewhere in the body. We also assume that there are no constants in the rules, except possibly for the query clause. Predicates that correspond to relations stored in the database (such as \textit{knows above}) are called \textit{EDB predicates} (extensional database predicates); those that are defined by rules (such as \textit{buys above}) are called \textit{IDB predicates} (intensional databases predicates). We make the standard assumption that no EDB predicate appears in the head of any rule.

Regular path programs are defined as follows:

1. Rules must correspond to productions of a regular grammar; in other words, the body of each rule must comprise either (a) a single EDB predicate and a single IDB predicate (an \textit{internal} rule), or (b) one or more EDB predicates (an \textit{exit} rule).

2. All IDB predicate occurrences must have the same number of arguments.

3. Given an internal rule of the form

\[
s(X,Y,U,V) : = e(X,Z,U), t(Z,Y,U,V).
\]

where \( s \) and \( t \) are IDB predicates, \( e \) and \( t \) have exactly one variable in common that does not appear in the head (\( Z \) above). The position of this variable in \( t \) is called the \textit{linking} position.

4. The position in the head \( s \) corresponding to the linking position in \( t \) is occupied by a variable (\( X \) above) which appears only in \( e \). This is the \textit{source} variable.

5. The remaining positions in \( s \) must be occupied by variables appearing in the same positions in \( t \)—the \textit{persistent} variables (\( Y \), \( U \) and \( V \) above).

6. The source (or linking) position for each IDB predicate in the program must be the same.

7. For the whole program there must be exactly one persistent position such that for every IDB predicate appearing in the body of a rule the variable occupying that position appears nowhere else in the body. This position is called the \textit{sink} position, and in each rule the corresponding variable is called the \textit{sink} variable.

\textbf{Example 2.3} Referring to the program in \textbf{Example 2.1}, it is easy to see that it conforms to the definition of a regular path program. In each of the internal rules, \( Z \) is the linking variable, \( X \) is the source variable, and \( Y \) is the sink variable. There are no other persistent variables.

Now turning to the first internal rule of \textbf{Example 2.2}, once again \( Z \) is the linking variable and \( X \) the source variable; hence, \( Y \), \( U \) and \( V \) are persistent variables. Because \( U \) also appears in \( e(X,Z,U) \), the third argument position in IDB predicates cannot be the sink position. From the second internal rule, we establish that the fourth IDB argument position (occupied by \( V \)) also cannot be the sink position. Hence, \( Y \) is the sink variable in each of the internal rules. \( \Box \)
The reader should not be mistaken into thinking that the EDB predicate must always precede the IDB predicate in an internal rule. The rule

\[ s(X, Y, V) \leftarrow t(X, Z, V), e(Z, Y, V). \]

where \( e \) is the EDB predicate, does not violate the definition: \( Z \) is the linking variable, \( Y \) the source variable, and \( X \) the sink variable. Some examples violating the definition are given below.

**Example 2.4** The pair of internal rules

\[
\begin{align*}
  s(X, Y, V) & : = e(X, Z, U), t(Z, Y, V). \\
  t(X, Y, V) & : = e(X, Z, V), s(X, Y, Z).
\end{align*}
\]

violates item (6) in the definition, since the source position is 1 in the first rule and 3 in the second. On the other hand, the pair of rules

\[
\begin{align*}
  s(X, Y, V) & : = e(X, Z, U), t(Z, Y, V). \\
  t(X, Y, V) & : = e(X, Z, Y), s(Z, Y, V).
\end{align*}
\]

violates item (7), since the first rule implies that the sink position must be 2, while the second rule implies it must be 3.

The definition of regular path programs can be extended in a number of ways. Firstly, we can allow a *chain* of EDB predicates of the form

\[ e_0(X, W_1), e_1(W_1, W_2), \ldots, e_n(W_n, Z) \]

rather than the single EDB predicate \( e(X, Z) \) in an internal rule. It is also possible to allow tuples of variables rather than single variables for source and sink variables. We do not use these extensions in the remainder of the paper as they tend to make the notation more difficult to follow.

In the next section, we consider regular path programs with queries in which constants appear, for example, \( ? - \text{buys}(john, Y) \) or \( ? - s(X, y_0, U, v_0) \). If the constant appears in the source position, we call the program *right-linear*; if constants appear in one or more of the persistent positions, the program is called *left-linear*. These terms are consistent with those used in [NRSU89a].

## 3 Argument Reduction

We now turn our attention to ways in which a left- or right-linear regular path program can be rewritten so that its evaluation can be performed more efficiently. In this section, we show that the techniques presented in [NRSU89a] can be extended to apply to these programs.

### 3.1 Left-Linear Programs

If any persistent variables are bound to constants in a query to a regular path program \( P \), we simply substitute the constants for the corresponding variables in all EDB predicates in \( P \) and delete the variables wherever else they appear in \( P \).
Example 3.1 Consider the program of Example 2.2 in which argument positions 2, 3 and 4 are persistent. If we have the query \( s(X, y_0, U, v_0) \), then the program is rewritten as follows.

\[
\begin{align*}
  s(X, U) & : - e(X, Z, U), t(Z, U). \\
  t(X, U) & : - e(X, Z, v_0), s(Z, U). \\
  s(X, U) & : - e(X, Z, U), e(Z, y_0, v_0).
\end{align*}
\]

All occurrences of \( Y \) and \( V \) in IDB predicates have been deleted, while all occurrences of \( Y \) and \( V \) in EDB predicates have been replaced by \( y_0 \) and \( v_0 \), respectively. As a result, the number of arguments in each IDB predicate occurrence has been reduced from four to two.

Given a regular path program \( P \), we can always rewrite it so that (1) each rule uses the same set of variables in its head, and (2) the source variable appears in the first argument position of each IDB predicate in the head, the sink variable appears in the second position, and the remaining persistent variables appear in the same order in each rule. We will assume this standard form from now on.

Let the query to the program be given by \( q(X, \overline{V}) \), the tuple of persistent variables being \( \overline{V} \). We assume that \( \overline{W} = W_1, \ldots, W_m \) is the subtuple of variables in \( \overline{V} \) that are bound to constants \( w_i, 1 \leq i \leq m \), in the query, and that \( \theta \) is the substitution that replaces each \( W_i \) by \( w_i, 1 \leq i \leq m \). Let \( \overline{V} - \overline{W} \) denote the removal of all variables in \( \overline{W} \) from \( \overline{V} \). The general method is as follows.

1. Given an exit rule of the form

\[
t(X, \overline{V}) : = \xi_1, \ldots, \xi_k.
\]

where \( \xi_1, \ldots, \xi_k \) are \( e \) (EDB) literals, transform it to

\[
t(X, \overline{V} - \overline{W}) : = \theta(\xi_1), \ldots, \theta(\xi_k).
\]

2. Given an internal rule of the form

\[
t(X, \overline{V}) : = e(X, Z, U), s(Z, \overline{V}).
\]

where \( U \) appears in \( \overline{V} \), transform it to

\[
t(X, \overline{V} - \overline{W}) : = e(X, Z, \theta(U)), s(Z, \overline{V} - \overline{W}).
\]

The query \( q(X, \overline{V} - \overline{W}) \) is now applied to the transformed program.

3.2 Right-Linear Programs

If the source variable \( X \) in a query \( q(X, Y, \overline{V}) \) is bound to a constant, we apply a transformation based on that of Magic Sets [BMSU86, BR87], similar to the technique in [NRSU89a]. The first step in such a transformation is the top-down propagation of the binding patterns through a program \( P \), leading to an adorned program \( P^{ad} \) [Ullm88], in which each IDB predicate \( p \) has an adornment \( \alpha \) indicating which arguments of \( p \) are bound and which are free. For example, \( p^{bf} \) means that the first argument of \( p \) is bound while the second is free.
Example 3.2 Consider again the program $P$ of Example 2.2. If we assume that the query is $\exists s(x_0, Y, U, V)$, then the adorned program $P_{ad}$ is as follows:

$$
\begin{align*}
    s^{bf}(X, Y, U, V) & : - e(X, Z, U), t^{bf}(Z, Y, U, V). \\
    t^{bf}(X, Y, U, V) & : - e(X, Z, V), s^{bf}(Z, Y, U, V). \\
    s^{bf}(X, Y, U, V) & : - e(X, Z, U), t^{bf}(Z, Y, U, V). \\
    t^{bf}(X, Y, U, V) & : - e(X, Z, V), s^{bf}(Z, Y, U, V). \\
    s^{bf}(X, Y, U, V) & : - e(X, Z, U), e(Z, Y, V). \\
    s^{bf}(X, Y, U, V) & : - e(X, Z, U), e(Z, Y, V).
\end{align*}
$$

In the query, only $X$ is bound so we start with the adornment $bf$ for $s$. Given that $X$ is bound in the first rule, by the time $t$ is evaluated in a top-down evaluation, $Z$ and $U$ will also be bound; hence the adornment $bf$ for $t$. This process continues until all adornments which are generated in a top-down manner for all IDB predicates have been considered.

The second step in the transformation is to use the standard technique to derive the set of magic rules for $P_{ad}$ [BMSU86]. The following example demonstrates the method.

Example 3.3 We use the adorned program $P_{ad}$ from the previous example. The magic rules are used to compute bottom-up those bindings for variables that would have been used in a top-down computation. These bindings are represented by so-called magic predicates, one for each adorned version of an IDB predicate in $P_{ad}$. The magic predicates are formed by prefixing $m.$ to the IDB predicates. We first generate a rule for the constant in the query:

$$
m_{-s^{bf}}(x_0)
$$

For each internal rule in $P_{ad}$ of the form

$$
t^{a1}(X, Y, \overline{V}) : - e(X, Z, U), s^{a2}(Z, Y, \overline{V}).
$$

we generate its magic rule by (i) prefixing both $s$ and $t$ with $m_{-}$, (ii) deleting all free variables in $s$ and $t$, and (iii) exchanging $m_{-}s$ and $m_{-}t$. The reason for step (ii) is that we are interested only in bound variables, while for step (iii) is that we want to simulate top-down evaluation by bottom-up evaluation. This gives rise to the following set of magic rules:

$$
\begin{align*}
    m_{-t^{bf}}(Z, U) & : - m_{-s^{bf}}(X), e(X, Z, U). \\
    m_{-s^{bf}}(Z, U, V) & : - m_{-t^{bf}}(X, U), e(X, Z, V). \\
    m_{-t^{bf}}(Z, U, V) & : - m_{-s^{bf}}(X, U, V), e(X, Z, U). \\
    m_{-s^{bf}}(Z, U, V) & : - m_{-t^{bf}}(X, U, V), e(X, Z, V).
\end{align*}
$$

In the final step of the transformation, magic predicates are introduced into the exit rules of $P_{ad}$. Unlike in the usual Magic Sets algorithm, only the magic rules and these modified exit rules are used to answer the original query.

Example 3.4 Continuing with our running example, we must substitute each of $m_{-s^{bf}}$ and $m_{-s^{bf}}$ into the exit rule of the program, giving:
\[ s(Y, U, V) : = \neg m_{\neg \text{aff}}(X), e(X, Z, U), e(Z, Y, V). \]
\[ s(Y, U, V) : = \neg m_{\neg \text{aff}}(X, U, V), e(X, Z, U), e(Z, Y, V). \]

Note once again that the number of arguments in all IDB predicates has been reduced by the transformation. \( \square \)

We now summarise the general method. Given a program \( P \) and query \( q(x_0, Y, V) \), \( P \) is rewritten as follows.

1. Generate the adorned program \( P^{ad} \) from \( P \) and the query.
2. Generate the magic rules from \( P^{ad} \) according to the algorithm in [BMSU86].
3. For each exit rule in \( P^{ad} \) of the form
   \[ p^\alpha(X, Y, V) : = \xi_1, \ldots, \xi_k. \]
   where \( \xi_1, \ldots, \xi_k \) are \( e \) (EDB) literals, generate the rule
   \[ q(Y, V) : = \neg m.p^\alpha(X, U), \xi_1, \ldots, \xi_k. \]
   where \( U \) is the tuple of persistent variables bound according to \( \alpha \).

The query \( q(Y, V) \) is now applied to the generated program.

In common with [NRSU89a], the above method has the advantage over Magic Sets that magic predicates are substituted into exit rules alone, the remaining rules of the original program being discarded.

4 Efficient Evaluation

The technique of argument reduction has been shown to speed up the evaluation of a range of Datalog programs. The programs to which the technique is applied in [NRSU89b] generalise the one-sided programs of [Naug87], the separable programs of [Naug88], and the right-, left- and combined-linear programs of [NRSU89a]. These programs are the so-called RLC-stable programs: those containing only right-linear, left-linear and combined-linear rules in terms of a single IDB predicate and one exit rule.\(^1\) For these programs it has been proved that the evaluation of a program \( P \) transformed by argument reduction is never less efficient than the evaluation of \( P \) transformed by the Magic Sets algorithm, which is the best general purpose technique for speeding up evaluation of Datalog programs. In fact, as shown in [NRSU89b], programs transformed by argument reduction can lead to an order of magnitude improvement over Magic Sets in terms of evaluation efficiency.

We have already seen that regular path programs can contain multiple exit rules (Example 2.1), as well as more than one IDB predicate and mutually recursive rules (Example 2.2). Thus, there are regular path programs that are not RLC-stable. On the other hand, regular path programs do not contain combined-linear rules, so there are RLC-stable programs that are not regular path programs.

We show below by means of an example that, given a regular path program that is not RLC-stable, the transformations described in this paper can also lead to an order of magnitude improvement over Magic Sets in terms of evaluation efficiency.

\(^1\)There are other restrictions as well.
Example 4.1 Once again, we consider the program $P$ of Example 2.2, excluding the persistent variables $U$ and $V$ which are not needed for the present purpose:

$$s(X,Y) : = e(X,Z), t(Z,Y).$$
$$t(X,Y) : = e(X,Z), s(Z,Y).$$
$$s(X,Y) : = e(X,Z), e(Z,Y).$$

Assume that the query is $\neg s(z_0,Y)$ and that the relation $e$ contains the $2n + 1$ tuples $\{(x_i, x_{i+1}) \mid 0 \leq i \leq 2n - 1\}$. The program $P^{ad}$ is as follows:

$$s^{ad}(X,Y) : = e(X,Z), t^{ad}(Z,Y).$$
$$t^{ad}(X,Y) : = e(X,Z), s^{ad}(Z,Y).$$
$$s^{ad}(X,Y) : = e(X,Z), e(Z,Y).$$

From $P^{ad}$ we get the following magic rules:

$$m \cdot s^{ad}(x_0).$$
$$m \cdot t^{ad}(Z) : = m \cdot s^{ad}(X), e(X,Z).$$
$$m \cdot s^{ad}(Z) : = m \cdot t^{ad}(X), e(X,Z).$$

The relation $m \cdot t^{ad}$ contains the $n$ values $\{x_1, x_3, \ldots, x_{2n-1}\}$, while $m \cdot s^{ad}$ contains the $n + 1$ values $\{x_0, x_2, \ldots, x_{2n}\}$. Substituting the magic predicate into the exit rule according to the method of Section 3.2 yields the following:

$$s^{ad}(Y) : = m \cdot s^{ad}(X), e(X,Z), e(Z,Y).$$

The relation $s^{ad}$ contains the values $\{x_2, x_4, \ldots, x_{2n}\}$, which is the answer to the query. On the other hand, the Magic Sets algorithm produces

$$s^{ad}(X,Y) : = m \cdot s^{ad}(X), e(X,Z), e(Z,Y).$$

where $s^{ad}$ contains the tuples $\{(x_{2i}, x_{2i+2}) \mid 0 \leq i \leq n - 1\}$. So in both cases the relation for $s^{ad}$ contains $n$ tuples. But now Magic Sets goes on to substitute magic predicates into the internal rules of the program as well, yielding:

$$s^{ad}(X,Y) : = m \cdot s^{ad}(X), e(X,Z), t^{ad}(Z,Y).$$
$$t^{ad}(X,Y) : = m \cdot t^{ad}(X), e(X,Z), s^{ad}(Z,Y).$$

Here $s^{ad}$ contains the tuples $\{(x_{2i}, x_{2j}) \mid 0 \leq i < j \leq n\}$, while $t^{ad}$ contains the tuples $\{(x_{2i-1}, x_{2j-1}) \mid 1 \leq i < j \leq n\}$. These two relations together contain exactly $n^2$ tuples, an order of magnitude larger than the relations in the program transformed by argument reduction. □

5 Conclusion

We have defined a class of Datalog programs, the regular path programs, whose efficiency of evaluation can often be improved significantly by applying the technique of argument reduction. This class of programs is incomparable to previous classes to which this technique has been applied.

It has been shown that bottom-up evaluation using the Magic Sets transformation followed by semi-naive evaluation is never less efficient than top-down evaluation [Ullm89]. On the other hand, Magic Sets does not perform as well as it might for certain classes
of programs and it is highly unlikely that any single method will prove to be the most efficient for all programs. This has led to the search for subclasses of programs which are amenable to specialised techniques for improving evaluation efficiency. One such class of programs are the RLC-stable programs [NRSU89b]. The regular path programs defined in this paper now provide another such class.

One obvious area of future research is to attempt to establish whether the class of regular path programs can be integrated with the RLC-stable programs, thereby producing a strictly broader class of programs to which argument reduction can be applied.

References


OBJECT ORIENTATION IN RELATIONAL DATABASES

M RENN HackKamp

Department of Computer Science
University of Stellenbosch

Abstract

There is, as yet, no single formal definition for the object-oriented approach. The object-oriented paradigm is described in terms of the concepts accentuated by the approach, namely classes, objects, methods, messages and class hierarchies. In view of these concepts, the object-oriented approach supports the properties of encapsulation, inheritance and polymorphism.

The object-oriented approach stands in contrast to many aspects of the relational data model traditionally used in databases. The data in a relational database is viewed as a semi-passive component, while the objects in the object-oriented approach are viewed as active components. The relational data model and the object-oriented approach also differ in many other respects.

Second generation relational database management systems do provide features which make them more object-oriented than their traditional first generation counterparts. In order to develop applications utilizing these systems, it may be necessary to consider object-oriented approaches. Ingres, as an example of a second generation relational database management system, provides some of the concepts of the object-oriented paradigm and it supports some of the properties of the object-oriented approach.

INTRODUCTION

The intent of the object-oriented approach is to provide a natural and straightforward way to describe real-world concepts and semantics. This approach allows the flexibility of expression necessary to capture the variable nature of the world being modelled and the dynamic ability to represent changing situations. A fundamental part of the naturalness of expression provided by the object-oriented approach is the ability to share data, code and definition.

At this stage, there is no single clear formal definition of object-orientation. It is a developing technology; there is as yet no agreement on the set of features and mechanisms belonging to object-orientation. It is usually just interpreted as an approach that exploits encapsulation (packaging). The object-oriented approach can, however, be described fairly accurately in terms of the concepts supported and the properties gained through this support.

The second generation relational database management systems (RDBMSs) provide features which make them more object-oriented than their traditional first generation counterparts. An example of such a second generation RDBMS is Ingres. With its Object Management Extension, it supports a degree of object-orientation. When developing methodologies for designing applications utilizing these systems [28], it is necessary to consider object-oriented approaches. It may be necessary to utilize object-oriented analysis and design approaches, should these systems prove to be substantially object-oriented.

The extent to which second generation RDBMSs provide for the concepts of the object-oriented paradigm and support of its properties are investigated in this paper. Ingres is used as the prime example, being more explicitly object-oriented than the other second generation RDBMSs. A few of the discussions refer to other RDBMSs as well. The first section addresses the object-oriented concepts and the second the properties of object-orientation. In the third
section, the second generation RDBMSs are evaluated with respect to the criticisms of the relational data model and the differences often cited between object-oriented systems and the relational data model.

1. CONCEPTS

The principle terms and concepts of the object-oriented approach include class, object, method, message and class hierarchy.

1.1 CLASS

The object-oriented approach involves classifying objects in classes according to the similarities between them. Every object is an instance of some class. A class consists of the following:

- **Name**: A symbol that identifies the class in representations or programs.
- **Set of instance variables**: A definition of the structure (fields or attributes) of all the objects of the class. Every instance variable has a name. The instance variable can be augmented with a type, which is the name of the class that determines the valid set of values for the instance variable [20,p193-194].
- **Set of methods**: A class, whether system- or user-defined, carries an understanding of the methods (operators) that can be applied to objects of the class. The methods defined for a class are very closely associated with the class. The instance variables of an object can only be modified indirectly by invoking the methods of its class [9,p689].

Thus, an object class specifies a name, a set of visible methods, a set of hidden instance variables and a set of hidden operations implementing the methods. An object has specific values for each of the instance variables defined in its object class. It shares the methods' operations with other instances of its class. The objects of a class often refer to other objects. In many systems the static properties of objects, as well as the references to other objects, are represented as instance variables. The value of an instance variable is an object itself, of a specified class [23,p5].

The objects of a class can be complex. A system usually provides a set of built-in primitive object classes, e.g. integer or float. A user can create simple object classes based on these primitive classes (see methods). Complex object classes can be constructed from combinations of existing object classes, which in turn may be complex.

Two cases of Ingres' support for object-orientation are considered in this paper. Case A addresses user-defined abstract data types (ADTs), while Case B addresses relations and views.

**Case A**: Through the Ingres Object Management Extension, it is possible for a user to define ADTs applicable to a particular application. For each ADT, a set of pre-specified functions, called SQL functions, have to be defined as well. These functions define the actions to be executed when the ADT is manipulated or accessed through a SQL data manipulation or query command. An Ingres user-defined ADT is analog to a class, with the attribute values assumed for the ADT as the object occurrences. An ADT, as an object class, consists of the following [16,p1.4-1.6,p3.5-3.39]:

- **Name**: The name of the data type is the name of the class.
- **Instance variables**: The component fields of the ADT are the instance variables of the class.
- **Methods**: The SQL functions defined for the ADT are the methods of the class.

**EXAMPLE**

Consider the definition of the ADT to represent ordered pairs:

```c
typedef struct _ORD_PAIR
{
    double op_x; /* x coordinate */
    double op_y; /* y coordinate */
} ORD_PAIR;
```

212
extern II_STATUS op_compare();
extern II_STATUS op_keybd();
extern II_STATUS op_getempty();
...
II_STATUS op_compare(scb, op1, op2, result);
II_DATA_VALUE *op1; kill
II_SCB *scb;
II_DATA_VALUE *op1;
II_DATA_VALUE *op2;
int *result
{
ORDPAIR *opv_1 = (ORDPAIR *) op1->da_data;
ORDPAIR *opv_2 = (ORDPAIR *) op1->da_data;
if = (opv_1->op_x > opv_2->op_x)
   { *result = 1; }
else if (opv_1->op_x < opv_2->op_x)
   { *result = -1; }
else if (opv_1->op_y > opv_2->op_y)
   { *result = 1; }
else if (opv_1->op_y < opv_2->op_y)
   { *result = -1; }
else
   { *result = 0; }
return(II_OK);
}
...

This definition is far from complete. It still requires a plethora of functions, as prescribed in the Ingres Object Management manual, to allow full SQL support. However, it illustrates that an ADT addresses the components of a class, for example:

- Name: The name of the ADT is ORDPAIR.
- The instance variables of the ADT are op_x and op_y.
- The methods of the ADT include the SQL functions op_compare(), op_keybd() and op_getempty().

Case B: A relation (table) structure is analogous to a class, with the tuples of the relation forming the object occurrences. The relation, as an object class, consists of the following [15,p2.34-2.38]:

- Name: The name of the relation is the name of the class.
- Instance variables: The attributes of the relation are the instance variables of the class.
- Methods: The procedures defined on the relation are the methods of the class.

It is necessary to consider the methods, as supported by Ingres, in more detail. In Ingres, it is possible to define database procedures associated with a relation. These procedures can be called through interactive interfaces and by applications to perform operations on the data contained in relations. All the necessary data manipulation operations can be implemented through procedures. Thus in Ingres, as an example of a second generation RDBMS, it is possible to associate procedures as methods with a relation as a class [15,p1.61-1.65],[33,p350-357],[34,p897-900],[35,p5-10]. The Ingres procedures are analyzed in more detail in the subsequent section on methods.

A relation, as a complex object, may be considered as a set of tuples, a tuple as a set of scalars and each scalar as an instance of a primitive built-in class or of a user-defined ADT. The components of a complex object are also called instance variables as they appear in every instance of the class of object [9,p688].

EXAMPLE

Consider the definition of a Ingres relation to document rivers occurring on a map, shown with two of its procedures:
Create table RIVER
( RIVERNAME  varchar(30) not null,
  COUNTRYNAME  varchar(30) not null,
  RIVERLENGTH  integer,
  STARTPOS    ORDPAIR,
  ENDPOS      ORDPAIR );

Create procedure RETRIEVE_RIVER_LENGTH
( RNAME  varchar(30) not null ) as
declare
  RLENGTH  integer not null;
begin
select RIVERLENGTH
  into :RLENGTH
from RIVER
  where RIVERNAME = :RNAME;
return :RLENGTH;
end;

Create procedure UPDATE_RIVER_LENGTH
( RNAME  varchar(30) not null,
  RLENGTH  integer ) as
begin
update RIVER
  set RIVERLENGTH = :RLENGTH
  where RIVERNAME = :RNAME;
end;

These definitions illustrate that a relation adheres to the description of class as follows:
- Name: The name of the class is RIVER.
- Instance variables: The instance variables of the RIVER class are RIVERNAME, COUNTRYNAME, RIVERLENGTH, STARTPOS and ENDPOS.
- Methods: Two of the methods defined for the RIVER class are the procedures named RETRIEVE_RIVER_LENGTH and UPDATE_RIVER_LENGTH.

Note that RIVER is a complex object class. Its instance variables refer to other classes, namely STARTPOS and ENDPOS of the ORDPAIR class, as well as COUNTRYNAME of the COUNTRY class.

1.2 OBJECT

In the object-oriented approach, all conceptual entities are modelled using a single concept, namely objects. An object belongs to a class and has a state (value). An object is an instantiation of its class. When a new instance of an object class is created, it has its own set of instance variables and it shares the methods of its class with other instances of its class [20,p193],[23,p5].

Every object should have identity as part of its structure. The identity of an object distinguishes it from all other objects. In most object-oriented systems, every object has a unique system-generated object-id, guaranteed never to change. The object-id can be considered as a hidden attribute; it is not visible to users, nor can be manipulated [9,p688].

Case B: A tuple of a relation can be considered as an object. A relational tuple contains as many instance variables (attributes) as the number of instance variables of its class (relation). Each tuple in a relation can be uniquely identified, usually by the values of the attributes participating in the relation's primary key [9,p688]. In some relational systems each tuple is uniquely identified by the RDBMS through a row identifier. In Ingres, the row identifier is specified by augmenting an attribute with the system_maintained clause. This system_maintained
attribute is system-generated and cannot be changed by a user. It can merely be accessed and used in cross references [15,p2.34-2.36].

EXAMPLE

Consider the following tuple as an example object of the RIVER class:

< "Nile", "Egypt", 4145, "(31,0)", "(32,32)" >

This object contains values for all the instance variables of the RIVER class. Assuming river names are unique, it is identified by the instance variable RIVERNAME. It can also be identified by an internal system_maintained attribute.

1.3 METHOD

The objects of each class have active properties specifying their allowed behavior. A method is an operation or a function which can be performed on objects of a particular class with which the method is associated. The set of methods defined for a class forms part of the definition of the class. The class (name) and its methods form the "public interface" for the objects of the class. The only way to operate on the objects of a class is through the methods defined for that class [9,p689].

Similar to built-in objects, there are usually a set of primitive methods for fundamental operations, eg scalar comparisons, arithmetic operations, string manipulation operations, etc. Some of the built-in methods are generic; they are understood by every class. In some systems, a generic function defines the public interface, while a method for each class supporting the interface is an implementation of that interface.

According to Moon, a method has five properties [22,p55]:

- A generic function that it specializes; thus a public interface supported by the method.
- An applicability condition, indicating when the method is appropriate when an object is passed as an argument to the generic function.
- Qualifiers that identify the method's role, where applicable.
- A parameter list that receives the arguments.
- The body of code executed when the method is called.

Case A: Where an ADT is considered as a class, the SQL functions defined for the ADT form the methods of the class. In the Ingres implementation, the instances of the ADTs can only be accessed and manipulated through the SQL functions defined as part of the ADT.

EXAMPLE

In the ORD_PAIR class defined in a previous example, the SQL functions op_compare(), op_keybld() and op_getempty() are some of the methods defined for the class.

Case B: Where a relation is considered as a class, the procedures defined on the relation would then be the methods defined for the objects of the class. As stated, the objects of the relation class would be the tuples of the relation. The tuples of the relation should then only be accessed and manipulated through the methods defined for the class. In the Ingres implementation it is possible to eliminate access to relations through the standard SQL data manipulation commands. This can be done through security control mechanisms; the activator of a procedure can have different security-based capabilities for the procedure and the relation.
EXAMPLE

The two procedures RETRIEVE_RIVER_LENGTH and UPDATE_RIVER_LENGTH defined in a previous example are examples of methods of the RIVER class.

The RETRIEVE_RIVER_LENGTH method satisfies Moon's characterization as follows:

- Generic function: The method specializes the generic SQL "select" operation.
- Applicability condition: The method is applied to a particular object, namely the instance (tuple) of the RIVER relation with the same RIVERNAME as identified in the parameter list.
- Qualifiers: The only qualifier applicable to this method is the "not null" clause on the RNAME parameter.
- Parameter list: The parameter list (RNAME, RLENGTH) defines the values interchanged between the method and the calling mechanism.
- Body: The body of the method is the "select" statement between the "begin" and "end" statements. It is not visible to the calling mechanism.

1.4 MESSAGE

Objects communicate and perform all computations via messages. A message is a specification to activate one of an object's methods. In order to apply a given method to a given object, it is necessary to send a message to the class of the object. The message is dispatched to the method by the class of the receiving object, to be applied to the indicated object. The behavior of an object is captured in the messages to which an object responds. The messages completely define the operational semantics of an object [9,p689],[13,p522],[22,p69].

A message to an object requests that it should carry out a specified operation defined as a method, but the message does not indicate how the operation should be performed. The receiving object class contains the method which specifies how the operation should be performed. The sender, presuming objects are intelligent, trusts the receiver to perform its request correctly. This "call-by-desire" notion is central to the object-oriented approach [13,p163],[22,p24]. When an object receives a message, it executes the method activated by the message and returns an object as a result to the sender [9,p689].

A message is a syntactic construct, usually consisting of three parts [13,p522]:

- An identification of the object singled out as the receiver of the message.
- The name of the method (operation) to be performed, also called the selector. It is merely a symbolic name for the actual operation to be performed.
- A list of arguments, namely additional objects or values passed as parameters.

The set of messages an object can respond to is called its protocol. The external view of an object is nothing more than its protocol [29,p7].

Case A: The activation of the SQL functions defined as part of an Ingres user-defined ADT can be considered a type of message. Whenever a data manipulation command is applied to a relation, the corresponding SQL functions are activated. The data manipulation message sent to the relation class causes a message to be sent to the underlying ADT class. The activated SQL function is the selector; it specifies the method to be executed. The affected object occurrences are specified by the SQL data manipulation command. The arguments are utilized as specified in the function definition.

Case B: The procedure call mechanism implemented in Ingres can also be considered a type of message. The procedure call explicitly names the selector through the name of the procedure. The exact objects to be affected by the message have to be identified through the value parameters passed to the procedure. The parameters can also be used to pass values utilized in the execution of the method associated with the procedure.

216
EXAMPLE

Consider the following activation of the UPDATE_RIVER_LENGTH procedure from the UPDATE_RIVER rule:

Create rule UPDATE_RIVER
   after update(RIVERLENGTH) of RIVER
   execute procedure UPDATE_RIVER_LENGTH
      ( RNNAME = new.RIVERNAMESPACE,
        RLENGTH = new.RIVERLENGTH );

This rule is fired whenever the RIVERLENGTH attribute of the RIVER relation is updated. It activates the UPDATE_RIVER_LENGTH procedure to perform the update. It can be considered as sending a message to the RIVER class to activate the UPDATE_RIVER_LENGTH method.

With a security-strict Ingres implementation, as described previously, users cannot access the tuples of relations through SQL data manipulation commands. The protocol of a class (relation) therefore only consists of the set of defined messages (procedure names).

1.5 CLASS HIERARCHY

Classes can be related hierarchically to each other. If class B is a subclass of class A, then every instance of B is automatically an instance of A, but the converse is not true. The class hierarchy captures the IS-A relationship between a class and its subclass. The objects of related classes share properties through inheritance. All subclasses of a class inherit all properties defined for the classes and can have additional properties local to them. The subclass is a specialization of the class and the class is a generalization of its subclasses. The implication of a class hierarchy is that if class B is a subclass of class A, an object of class B may be used wherever an object of class A may be used. Subclass B of class A will automatically inherit features from A [1,p4-6],[14,p21]. Inheritance is discussed in more detail in a subsequent section, under the properties of the object-oriented approach.

Case A: The user-defined ADTs of Ingres do not support class hierarchies. An instance of one ADT cannot be an instance of another ADT. One ADT can be used in the definition another ADT, but that forms an IS-PART-OF relationship encountered in complex objects and not an IS-A relationship.

Case B: Relation B forms a subclass of another relation A if the set of tuples of B form a subset of the set of tuples of A. This is typically implemented through select views in second generation RDBMSs. The set of tuples visible through a view B, defined on a relation or view A, form a horizontal subset of the set of tuples visible through A.

EXAMPLE

In Ingres, a view is defined as a virtual relation, based on a "select" retrieval from either a base relation or another view. Thus, a subclass LONGRIVER can be defined for the class RIVER:

Create view LONGRIVER as
   select RIVERNAMESPACE, COUNTRYNAMESPACE, RIVERLENGTH,
       STARTPOS, ENDPOS
   from RIVER
   where RIVERLENGTH > 500
   with check option;

This LONGRIVER subclass (view) inherits all the specified instance variables (attributes) from the RIVER class (relation). Note the attributes are not automatically inherited; they have to be specified. Only those objects (tuples) with a RIVERLENGTH longer than 500 will be instances of the subclass, but all instances of LONGRIVER will also be instances of RIVER.
A view is a virtual relation. This is particularly valid in some other second generation RDBMSs, such as PowerHouse StarBase, where fully updatable views can be defined [6]. In such systems, views can be used in all data manipulation and data definition commands where relations can be used. This is not valid for all Ingres views.

2. PROPERTIES

The important properties of the object-oriented approach can be classified under encapsulation, inheritance and polymorphism.

2.1 ENCAPSULATION

Central to the object-oriented approach is the notion that the objects of interest in the real world can be modelled most effectively through recording them together with the operations that are permitted on them [3,p41]. Encapsulation enables the restriction of the effects of change on software. It provides a strong form of data independence. By including the methods with the definition of an object class and by only providing access to the objects of the class through the methods, it hides implementation and representation details from users. This allows the representation details to be changed, if necessary, in a controlled manner without affecting applications. This also enforces disciplined access to objects, only through the public interfaces provided by the methods defined for the class of the object [9,p689-700], [24,p263].

Case A: The implementation of ADTs in Ingres provides a high degree of encapsulation. The SQL functions applicable to an ADT are defined and recorded together with the structure of the ADT. The instances of the ADTs can only be manipulated through the SQL functions encapsulated with the ADT.

Due to its high degree of encapsulation, the ADTs as implemented in Ingres, provide a high degree of data independence. The actual storage structures can be changed. By changing the corresponding SQL functions, the applications utilizing the ADTs can remain unaffected by the changes.

Case B: The Ingres relations and views, if considered as object-oriented, only support a limited degree of encapsulation. It has been illustrated that classes can be defined, with methods inherent to the class. If not properly controlled, the objects (tuples), as instances of the classes (relations or views), can also be accessed through other mechanisms than just the methods (procedures) associated with the classes. The tuples can also be accessed through standard, generic SQL data manipulation commands.

In the Ingres implementation, if all applications only access the data stored in the database through procedures, a strong form of data independence can be achieved. This data independence could be improved even more through the use of updatable views, not presently available in Ingres. The representations of the objects (tuples) of a class could then be changed. By changing the corresponding procedures, the applications could remain unimpaired.

2.2 INHERITANCE

Inheritance is the concept used in object-oriented approaches to define objects that are almost like other objects, possibly with a few incremental changes. Inheritance makes it possible to declare that certain specifications are shared by multiple parts of a system. Inheritance allows classes to pass properties to each other. A subclass may share or inherit various characteristics of its superclass. These characteristics usually include the storage representation of the supertypes and the methods provided by the supertype [31,p188].

The advantage of inheritance is that it reduces the need to specify redundant information and it simplifies updating. Information about many object instances can be changed and entered in a single update action [13,p523].

Case A: Because the Ingres ADTs do not support class hierarchies, they do not provide inheritance properties either.
Case B: The implementation of updatable views in second generation RDBMSs such as Ingres provide a high degree of inheritance. It is standard procedure to define a view which is almost like a relation or almost like another view. A view defined on a base relation or another view also inherits a number of properties of the base relation or the view, as described subsequently. With a view being a virtual relation, merely a materialization of its underlying base relation, it shares many properties of the base relation.

2.2.1 Types of inheritance

Two types of inheritance have been identified, namely structural inheritance and behavioral inheritance [9,p690]:

- With structural inheritance subclass B automatically inherits all the instance variables of class A. Class B might have additional instance variables of its own which A does not have, as described in the next section on property modification.
- With behavioral inheritance subclass B automatically inherits all the methods that apply to class A. Class B might have additional methods of its own which do not apply to A, as described in the next section.

Case B: In relational views, the following inheritances are applicable:

- Structural inheritance: View B only inherits the attributes of A included in its definition. It can only define limited new attributes, namely constants and attributes derived from the attributes of A.
- Behavioral inheritance: Procedures cannot be defined for views, thus a view cannot inherit methods.

2.2.2 Property modification

The characteristics of a class inherited by a subclass may be changed or overridden in the subclass. A subclass may re-specify the storage representation or give new definitions for methods inherited from the supertype [14,p12],[24,p263]. A specialized class can modify the properties inherited from its superclass through additions and / or substitution [31,p189]:

- Addition allows the introduction of new variables, properties or methods in a class, which do not appear in one of its superclasses in the hierarchy.
- Substitution (or overriding) is the specification of a new value of a variable or property, or a new method for a selector that already appears in some superclass. All descriptions in a class, namely variables, properties and methods, are inherited by a subclass unless overridden in the subclass.

Case B: In relational views, class modification is supported as follows:

- Addition: A view can have additional attributes (instance variables), for example derived attributes. It cannot have additional procedures (methods) only applicable to the view.
- Substitution: An attribute can be renamed in a view, thus a derived attribute can be renamed to override its base relation definition. Methods cannot be applied to a view.

2.2.3 Source of inheritance

The source of inheritance indicates the superclasses from which a class inherits its properties. Class inheritance can be classified under single inheritance, multiple inheritance and partial inheritance [23,p7-8]. For both single and multiple inheritance, the following rule applies: All properties of a class, namely its instance variables and methods, are inherited by a subclass, unless overridden in the subclass. It should be possible to override inherited methods, eg a display method may have to display additional instance variables [13,p523].

With single inheritance, a subclass may inherit instance variables and methods of a single parent class, possibly adding some instance variables and methods of its own [23,p7].

Case B: Single inheritance occurs in a select view or in a view exactly mapped on a single base relation. The LONGRIVER view illustrated previously is an example of a subclass with single inheritance.
In many object-oriented systems, class hierarchies are essentially hierarchical in nature, thus only supporting simple inheritance. There is no direct support for associations, e.g. many-to-many relationships [9,p689]. In other systems, a class can have more than one superclass. Through multiple inheritance a subclass can then inherit properties from more than one class. Multiple inheritance is a natural extension to single inheritance, where a union of instance variables and methods are inherited from multiple parent classes. Then the class hierarchy is generalized to a class lattice, as represented by a directed acyclic graph structure. The class lattice often simplifies data modelling and requires fewer classes to be specified than required with a class hierarchy [1,p7],[23,p7].

Name clashes can occur in the presence of multiple class inheritance. Two similarly named, but differently implemented methods can be inherited from two parent classes. The same applies to the inheritance of identically named instance variables. In new methods such clashes can be distinguished by prefixing the inherited method or instance variable with the name of the parent. However, such clashes are most pronounced when class hierarchies are changed. It must be considered how the changes to the definition or implementation of a class will affect inheriting subclasses. There are two types of name clashes that can occur in a class lattice. In most systems these naming conflicts are resolved by precedence:

- A naming conflict can occur between a class and its superclass. This can also happen in a class hierarchy. The precedence is usually given to the definition of the class, overriding the definition of the superclass.
- A naming conflict can occur between the properties of the superclasses of a class. The superclasses of a class are usually represented as a list. The precedence is usually given to the definitions of the superclasses occurring earlier in the list. Alternatively, conflicting properties may be explicitly renamed in the definition.

Case B: Consider a join view, containing all the attributes of the underlying base relations. It is the typical example of a subclass supporting multiple inheritance. The view inherits the attributes from both the underlying base relations. In the Ingres implementation, as in most other relational systems, naming conflicts have to be explicitly resolved in the view definition. The context of attributes occurring in both underlying base relations have to be indicated, by prefixing the attribute name with the relation name in the view definition.

EXAMPLE

Consider the relation RIVER shown in a previous example, together with the following relation:

COUNTRY (COUNTRYNAME, CONTINENT).

The following view has multiple inheritance properties:

Create view RIVERLOCATION as
   select RIVERNAME, RIVER.COUNTRYNAME, CONTINENT,
   RIVERLENGTH, STARTPOS, ENDPoS
   from RIVER, COUNTRY
   where RIVER.COUNTRYNAME = COUNTRY.COUNTRYNAME;

Note the explicit name resolution by augmenting the COUNTRYNAME attribute name with the RIVER relation name. COUNTRYNAME occurs in both the RIVER and CONTINENT relations. Note such a join view is not updatable in Ingres.

Partial inheritance is when some properties are inherited and others are suppressed. In such a case neither class is a subclass of the other, but they are undeniably related. Partial inheritance is convenient for aspects like code sharing, but it can create a mess in a class hierarchy [23,p8].

Case B: A project-join view is the typical example of a subclass with multiple, but partial inheritance. Although the view inherits properties from both its underlying base relations, it does not inherit all the properties, as some attributes can be eliminated through the project part of the specification. Similar to the problems encountered in a class hierarchy or class lattice, a non-key preserving project-join view can introduce many problems. One such problem is that view may not necessarily be updatable [7,chap 17].

220
2.3 POLYMORPHISM

In general, polymorphism means "having or assuming different forms". In the context of object-orientation, polymorphism refers to the capability of different classes of objects to respond to exactly the same message protocols. The protocols enable a system to treat objects that arise from different classes uniformly. Message protocols extend the notion of modularity (reusability) to polymorphism (interchangeability) [31,p183]. Polymorphism is closely related to class inheritance. The same operations that apply to a parent class also apply to instances of its subclasses [23,p10].

The term overloading is also used for polymorphism. Operator overloading means that the same operator symbols can be used for the same operations or different data types. Thus, distinct methods can be given the same name for distinct classes. Overloading is performed when the definition of a method that applies to class A is extended or even totally changed to perform additional or totally different operations when applied to an object of a subclass B of class A. The operator is therefore overloaded and its true meaning can only be resolved by looking at its operand types in the context of the class where overloading is applied [9,p690],[13,p522].

A polymorphic function can be applied uniformly to a variety of objects. The "same" operation maintains its behavior transparently for different argument types. With ad hoc polymorphism or "mere" overloading, two operations coincidentally share the same name, but otherwise have completely different behaviors [23,p10].

Case A: The user-defined ADTs defined in Ingres support polymorphism to a large extent. For every ADT, the same SQL functions are defined. These "same" functions behave similarly; they perform the same operations on the ADT, but the implementation of a function for each different ADT can differ substantially. Thus the behavior (execution) of the method (SQL function) is determined by the context of its class (ADT).

Case B: The procedures used with relations do not support polymorphism at all. This is due to the fact that each procedure is required to have a distinct name.

3. EVALUATION

It has been stated that object-oriented databases have advantages over databases based on the traditional data models [13,p524-525]. This section evaluates the object-orientedness of the second generation RDBMSs with respect to the criticisms often cited against the relational data model, as well as in terms of the major differences usually cited between the object-oriented approaches and the relational data model.

3.1 PASSIVE VS ACTIVE

The main difference between a traditional (relational) DBMS and an object-oriented DBMS is in the passive and active behavior of the underlying database. A traditional database is passive, it embodies a collection of structured data, such as data contained in relations. The data to be processed and manipulated are accessed by applications via the DBMS. Some of the data are temporarily stored and processed in application program data structures; the results of the actions are often written back to the database. In the object-oriented approach, the database contains objects that are made up of both passive data objects and active methods to reflect the behavior of the data. Once a user request is transmitted to the object base, the objects respond to the request in a way consistent with their defined behavior. The methods take the necessary action and invoke other methods through messages to complete the request. The entire request is completely performed within the object base [24,p262-263].

In most of the second generation RDBMSs, for example Ingres, Sybase and StarBase, dynamic properties are considered part of the database. In these systems, functionality in the form of procedures, rules and triggers are defined as part of the database. These active components are automatically activated and executed when predefined conditions occur [27,p183-200].
3.2 APPLICATION DOMAIN

The simplicity of the relational data model is dependent on its use in modelling applications where the data is confined to a small number of different types of data related in well-defined ways. Its use can be limited when applied to complex, highly structured application domains [3,p39].

With the advent of advanced applications like engineering, design, office or knowledge-based systems, stored entities tend to show very complex internal structures. These structures may overlap and may comprise large numbers of possibly substructured, properties. Relational systems impose the first normal form constraint. This means that the object space must be mapped into a collection of flat relations. Using relational record-oriented data models for the applications listed above result in awkward database designs and complex retrieval operations. With this approach, much of the inherent semantics of complex object composition is lost. Foreign key joins have to be performed to reconstruct complex objects. It has been stated that the simple relational record structure, with atomic attributes, can be considered a major disadvantage of relational systems. It is difficult to express the semantics of complex objects in this fashion [3,p39],[12,p54],[17,p275]. Various solutions to these problems have been proposed. Some current approaches preserve the relational model [36], some add new features including user-defined types and procedures [32], while others abandon the relational model in favour of object-oriented approaches [1],[32]. While some parties state that a relational system can be made object-oriented, others state that object-orientation, including object identity and encapsulation, cannot co-exist with the declarative access to data and flexibility found in RDBMSs [11,p179].

The second generation RDBMSs already support many more object-oriented concepts and have many more of the properties associated with object-oriented systems than their first generation counterparts. The extent of this object-orientation is discussed in the conclusion. In addition to object-orientation per se, these systems support much more advanced data types. While Ingres supports ADTs [16], systems like Informix and StarBase support BLOBs and other multi-media data types [6]. With these data types, it is possible to store the data typically encountered in the application domains mentioned above. These include text, graphical and voice images, which can easily be accommodated in BLOBs.

3.3 INTEGRITY SEMANTICS

There seems to be controversy regarding the support of semantics related to integrity control.

Advocates of the object-oriented approaches state that it is impossible to capture and control much of the semantics related to the integrity control of a complex application with such a simple framework as the basic relational model. They state that although the relational model enforces referential integrity, it has no mechanism for distinguishing and enforcing the different types of relationship that may exist between entities, such as association (many-to-many), designation (one-to-many) and characterization (one-to-many existence dependent) links. If such a distinction were to be made, it would be possible to define the semantics of operations to create and delete relationships differently for each case [3,p39].

Garey & Jackson reiterated the point by stating that the object-oriented "data model" differs from the relational data model in the following respects [13,p525]:

- In the relational data model tuples in a relation are explicitly identified through primary key attributes. In the object-oriented approaches, an internal identifier is assigned to each object occurrence. Relationships are implemented using the internal identifiers. Any key changes are immediately visible to all objects, while in the relational model primary key updates have to be propagated to all foreign keys.

- In the object-oriented approaches, referential integrity can be achieved using the hidden identifiers, as relationships are explicitly defined. In the relational systems relationships are implicitly defined through foreign keys.

- An object-oriented schema captures integrity constraints, by recording properties of the entities in the system in a data dictionary. Object methods can be viewed as functions, allowing a more precise notation for specifying constraints.
These statements are not valid in view of the extensive facilities provided by second generation RDBMSs. For example, in the Database Application Development approach, the different types of relationships, with more semantics than mentioned above, can be modelled and can eventually be incorporated in the database definition. The integrity controls associated with these relationship types are then continuously and automatically enforced by the RDBMS [28,p103-213].

Advocates of the "pure" relational data model state that the object-oriented approach has shortcomings related to integrity control. The object-oriented user effectively has to understand three forms of different "referential integrity" control mechanisms [9,p702-703]:
- Embedding object B within object A.
- The traditional foreign key mechanism, through messages and methods.
- The class hierarchy concept.

They state that the implications of choosing one of these three representations over another is not clear. The asymmetry that such considerations give rise to is unfortunate. Even more, the object-oriented approach does not provide convenient declarative means of stating the referential integrity constraints or the actions to be performed on the violation of the constraints. For example, the operation corresponding to cascading delete does not even exist.

### 3.4 REAL WORLD SEMANTICS

It has been stated that due to the rigid framework of traditional data models, a semantic gap between the modelled world and its database representation usually cannot be avoided. It is impossible to represent all the semantics of interest in the database. The "remainder" has to be captured by the application programs using the database or by the interpretation of the result of database queries by the user self [12,p53].

The same comments apply here as stated under integrity semantics. Numerous application semantics, specifically related to data relationships, constraints and active semantics control can be modelled during database application design. These can then be incorporated in the database definition, where they are automatically activated and enforced [28, 103-213].

### 3.5 SCHEMA CHANGES

Banerjee et al stated that most existing relational systems allow only a few types of schema changes. They only allow the dynamic creation and deletion of relations (classes) and the addition of new columns (instance variables) in a relation. This is because the conventional record-oriented business applications they support do not require more than a few types of schema changes; also the data models they support are not as rich as object-oriented data models [1,p10-11].

In the object-oriented approaches, the database schema consists of the class definitions and the inheritance structure of the class lattice. Schema evolution is the ability to dynamically make changes to this schema. Banerjee et al classified schema changes according to a complete taxonomy, based on the representation of a class lattice as a graph. The essentials of the taxonomy is that it should be possible to add, drop and change classes, instance variables (descriptors and relationships), methods and inheritance [1,p10-16].

The set-oriented paradigm of the relational data model is exploited in RDBMSs to provide adaptability. The RDBMSs, for example Ingres, allow considerable schema and view changes. Columns can be added or deleted from relations (both descriptive attributes and foreign keys). Procedures can also be added and deleted from the database schema [15,p2.67-2.71].

Advocates of the relational data model state that although the object-oriented schemas provides support for version management and changes in databases, it has schema related problems. The mentioned drawbacks of the object-oriented approach, in contrast to the traditional data models, are the following [13,p525]:
- The object-oriented data models are more difficult to implement.
The object-oriented schemas are more rigid. A schema may be extended easily, but it is difficult to restructure defined relationships. The relational model places few constraints on the extension and modification of relational schemas.

- Resolving name conflicts resulting from multiple inheritance is more difficult with schema evolution, where the existing instances of the modified classes and subclasses must be preserved.

### 3.7 IMPEDANCE MISMATCH

Object-oriented databases alleviate the impedance mismatch between programming languages and DBMSs. In conventional database systems there is a mismatch between conventional record-at-a-time third generation programming languages and the set-at-a-time data manipulation language supported by the RDBMSs. Object-oriented databases are more complete in that they provide the necessary expressive power to perform the computations of an application through object-oriented programming languages [17,p274-275].

In some of the languages of the second generation RDBMSs, this impedance mismatch is reduced considerably. For example, the GDML language used in StarBase has built-in looping and set handling constructs. GDML can be used in a standard programming language such as C, Cobol or Pascal, to process a set of relational tuples without having to utilize a cursor [5,p2.7-2.12].

### 3.7 QUERIES

There will always be the need to access data in unforeseen ways for ad hoc queries. Object-oriented systems seem too rigid in only allowing operations through predefined methods [9,p701]. In addition, many of the object-oriented systems do not include the notion of "the set of all instances of a class", which is fundamental in database management systems [4,p17,p30],[9,p701].

Accessing data by means of user-defined methods, instead of (standard) system-supplied methods seems contradictory to the objective of optimization. The record-at-a-time flavor of object-oriented systems means that query optimization as found in relational (set-at-a-time) systems is impossible. To quote Date, "the record-at-a-time approach of object-oriented programs is a throwback to pre-relational systems". The notion of an object-oriented query language does not seem viable either. Indexing in an object-oriented system seems even more complex [9,p702-704].

### CONCLUSION

This paper documents the results of an investigation into the object-orientation supported by second generation RDBMSs, as the background for developing application design methodologies for these systems. In particular, it addresses the ADTs, relations and views as implemented in Ingres. These results are summarized in the following tables:

<table>
<thead>
<tr>
<th>Concept</th>
<th>ADT</th>
<th>Relations &amp; Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>Object</td>
<td>Partly</td>
<td>Supported</td>
</tr>
<tr>
<td>Method</td>
<td>Supported</td>
<td>Implicitly</td>
</tr>
<tr>
<td>Message</td>
<td>Partly</td>
<td></td>
</tr>
<tr>
<td>Class hierarchy</td>
<td>Not supported</td>
<td>Read only</td>
</tr>
</tbody>
</table>

224
<table>
<thead>
<tr>
<th>Property</th>
<th>ADT</th>
<th>Relations &amp; Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation</td>
<td>High</td>
<td>Limited¹</td>
</tr>
<tr>
<td>Data independence</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Inheritance</td>
<td>None</td>
<td>Limited</td>
</tr>
<tr>
<td>Structural</td>
<td>None</td>
<td>As defined</td>
</tr>
<tr>
<td>Behavioral</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Addition</td>
<td>None</td>
<td>Limited²</td>
</tr>
<tr>
<td>Substitution</td>
<td>None</td>
<td>Limited</td>
</tr>
<tr>
<td>Single</td>
<td>None</td>
<td>Limited</td>
</tr>
<tr>
<td>Multiple</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Partial</td>
<td>None</td>
<td>High</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>High</td>
<td>None</td>
</tr>
</tbody>
</table>

The following notes are applicable to the tables:

1. Values are validated through functions defined for domain integrity, but occurrences are not uniquely identified. There can be many occurrences of the same object.
2. Objects are not explicitly identified as the targets of messages. The objects affected by a message are implied by the context of the message in terms of a higher class message, namely the SQL data manipulation language command.
3. Encapsulation applied to a relation or view is dependent on proper security enforcement.
4. Views and relations have limited instance variable addition; they can only add attributes derived from the attributes of an underlying base relation or view.

Thus the second generation RDBMSs, such as Ingres, support a reasonable degree of object-orientation. This indicates that aspects of object-orientation could be considered when developing methodologies for designing applications targeted for these systems. There are, however, many object-oriented application analysis and design approaches; only a limited number of these seem appropriate. According to the results presented in this paper, approaches which stress the object-oriented concepts, together with encapsulation, should be considered. Approaches based on or stressing inheritance and polymorphism may not be applicable.

References


[27] MH Rennhackkamp, Relational Database Triggers, In [19].


[29] D Robson, Object-Oriented Software Systems, In [25].


Building a secure database using self-protecting objects

M S Olivier and S H von Solms
Department of Computer Science
Rand Afrikaans University
Johannesburg

May 23, 1991

Abstract

In current database systems the responsibility for enforcing security is often given to the various application programs. Even where the Database Management System (DBMS) does supply security mechanisms, a single application program often handles sensitive transactions for some users and therefore needs a high clearance for accessing data—this may render the provided mechanisms inadequate. Furthermore, the user's identity is often concealed because the user has many 'software agents' acting on its behalf—especially in distributed environments. A simple mapping between subjects and objects is no longer possible.

We propose a model for extending Object-Oriented Database Systems to enable objects themselves to ensure security—ie to protect themselves. This extension is based on the concept of 'baggage'—baggage is collected from all components involved in any request; this baggage may then be verified by the object against its personal security profile before any method is executed.

Keywords: Security, Multilevel Secure Database, DBMS, Object-oriented, Path Context Model (PCM)

1 Introduction

This paper describes a model to build a secure object-oriented database based on the Path Context Model (PCM) [2,3,4]. We will refer to this model as SECDB (SECure DataBase).

Both the appeal and concerns regarding security of object-oriented databases are discussed in [11]. On the positive side object-orientation supports encapsulation, the possibility to model security in real-world terms and the potential of inheritance. On the negative side the main problem occurs when a class and its superclass have different sensitivity levels: this may allow information to flow from a higher sensitivity level to a lower level.

The necessary extensions to an object-oriented database to support security are considered. Such extensions must fit as cleanly as possible into the object-oriented model. The proposed mechanisms are designed to be objects themselves, conforming to the normal object, class and inheritance structures. Also, in line with the object-oriented database model, the security information must be stored as part of the database, together with the data and code.

We will mainly be concerned with the secrecy aspect of security, although the proposed model has some advantages regarding integrity.

The reader is referred to [8] for a discussion of the classical secure database implementation strategies.

*Address correspondence to Martin Olivier, Department of Computer Science, Rand Afrikaans University, PO Box 22,
Johannesburg, 2000 South Africa; Email: Martin.Olivier@f1.n7101.zs.fidonet.org

228
2 Object-oriented extensions

This section introduces the terminology used. The concepts are treated in more detail in subsequent sections.

2.1 Object-oriented basics

We consider objects, classes, inheritance, methods and messages as the necessary ingredients of object-orientation [14, pp 8–11]. Also, in a secure system encapsulation [14, p 11] must be enforced—that is an object’s instance variables must be hidden from all methods except the object’s own methods.

2.2 Messages and baggage

Throughout this paper we use the term ‘message’ to refer to an active message: it starts to exist when it is sent (by a user or from some method which is being executed). We will say a message ‘activates’ a method; messages sent by the executing method are ‘spawned’ by the original message: the message directly responsible for sending a message will be referred to as the ‘spawning’ message.

Example

![Diagram](image)

Figure 1: Messages illustrated

See figure 1. Suppose a user sends the message GetCost to an object ProjectX. The method GetCost sends the message GetRate to the object Worker and then GetDate to the object Clock. We say that GetCost spawned GetRate and GetDate. The spawning message of both GetRate and GetDate is GetCost. Note that the message GetCost starts to exist when it is sent by the user; GetRate exists until it returns an answer. After that GetDate starts to exist and ends its existence.
when it returns the date. After this the *GetCost* message stops to exist when it returns its answer.

Messages are required to carry ‘baggage’ with them. SECDB is designed to support the baggage concept as defined in the Path Context Model (PCM) [2,3,4]. Informally, PCM requires that ‘baggage’ or information must be collected about all relevant software and hardware components involved in the handling of any request. A profile is associated with every item in the system to which access must be controlled—this profile specifies which components are required to take part in the handling of the request and which components are not allowed to take part. A profile may, for example, be defined to

1. Allow access only if one of a few specified users has originated the request; but

2. Deny access if some non-encrypted communications line has been used to transmit the request.

We expand on this later; initially this baggage may be viewed as the ‘clearance level’ of the sender or originator of the message: it is an indication of the trust associated with the request.

### 2.3 Profile objects

SECDB defines a profile as an object containing security information about protected information; given the baggage carried by a message, a profile can use its security information to determine whether the message may be allowed to access the protected information. The security information will typically be allowing and prohibiting contexts as defined in PCM; initially this information may be viewed as the ‘sensitivity level’ or ‘classification level’ of the information protected by the specific profile—ie a restriction of who is allowed to access the information. Profiles are normal objects—instances of profile classes—placing no additional requirements on the underlying system.

Since messages carry information they need protection and will therefore also have profiles associated with them. The information carried by a message will typically consist of parameters; however, the information can be much more subtle: the fact that a message has been sent can compromise information. Note that messages are active entities which cannot be accessed (assuming that the operating system protects executing programs). Messages therefore do not pose a security threat—the problem arises when such a message ‘drops’ information somewhere where it can be accessed, eg by assigning it to a variable or by creating an object. So, whenever a message is sent, profiles are taken along for all the information contained in it—contained directly in its parameters or contained implicitly. These profiles carried by messages will then be associated with any objects created or modified by that message. The profiles associated with a message will also be tagged to messages spawned by it. This will ensure that the *-property [1] is not violated when messages are sent between objects; simply put, this property requires that no user (or application) can write information he has access to where someone without the proper authorization can then read it. We will formalize this mechanism later. See section 5 for an example.

It is important to distinguish between baggage carried by messages and profiles carried by messages. Baggage is a record of the path a request has followed, ie which user initiated the request, which application programs were involved, which networks have taken part, etc. When a message arrives at an object, the object’s profile(s) will look at this baggage to determine whether access should be granted to the message. On the other hand, a message also carries profiles with it to protect the information contained by the message. These profiles will have no influence on what the message may or may not access; the profiles will rather be attached to any variables modified or objects created by the message, controlling access to those variables and objects.

### 2.4 Gates

In SECDB a gate is a ‘boundary’ around every object in the system: whenever a message is sent to an object, the message will be intercepted by the sending object’s gate and then by the receiving object’s gate. The sending object’s gate will tag any profiles associated with the sending object.
to the message. The receiving object’s gate will ask access permission from the receiving object’s profile(s). The profile will make this decision based on the baggage sent with the message and its internal profile information. If access is denied, it is the gate’s responsibility to handle the rejection of the message. If access is allowed, the gate will associate any profiles carried by the message with the instance variables changed or created in the receiving object (in addition to any existing profiles).

3 Profiles

3.1 Description of profiles

Profiles will be defined for information in a SECDB database to which access is restricted. Profiles are objects containing access requirements: these requirements will typically consist of a list of objects (users, programs, etc) that may access the information and a list of objects that may not take part in a request to access the information. Based on the baggage carried by a message and these requirements in a profile the profile can determine whether the message satisfies the access requirements or not. PCM specifies that the access requirements must be in the form of a Random Context Grammar [13]: it will specify which contexts (ie human, software and hardware components) are required to be in the baggage and which contexts are not allowed to appear in the baggage.

The profile may also support methods to dynamically modify the profile data; however, if such mechanisms are too general, the resulting security may be too complex, reducing trust. We do not consider meaningful restrictions to methods supported by profiles in the current study.

SECDB additionally requires that a profile object provides a method GrantAccess, which may be called by the protected object’s gate and which will reply with a Yes or a No.

Note that the PCM requirement of a Random Context Grammar profile may be modified to yield other interesting versions of SECDB; we discuss one such possibility later.

3.2 Profiles as objects

The implications of profiles being objects are

1. There are profile classes, subclasses and superclasses; and
2. Profile objects are protected by profiles themselves.

Considering (1), it is convenient to assume that a profile class has more restrictive access requirements than its (profile) superclass, i.e. that each subclass has a higher (or equal) ‘sensitivity level’ than its superclass. A hierarchy of profile classes then forms a partially ordered set of ‘sensitivity levels’. In order to guarantee this, SECDB requires that GrantAccess in any profile subclass will send a message to GrantAccess in the superclass of the profile subclass; access will be denied if it is denied by the superclass.

From (2) it follows that there is potentially an infinite chain of profiles. This problem is easily (and soundly) solved by allowing one profile object to act as its own security profile—the problem is similar to the one caused by the fact that classes are indeed objects and therefore instances of some metaclass, which is again an object itself.

3.3 Associating profiles with information

In order to protect information, profiles are specified when a class is defined. The specified profile(s) will then be used for all instantiations of that class.

An object may be viewed as a layered entity: The outermost layer contains those instance variables and methods (and shares those class variables) defined in its class; the next layer those defined in the superclass of its class, and so on up to the innermost level which contains the items
defined highest up in the class hierarchy. (In Smalltalk terminology [5], it contains the class variables and methods defined in class Object.) Figure 2 illustrates this. Since information at different levels in the class hierarchy may have different security requirements, different layers of an object may be protected by different profiles.

![Class hierarchy and Instantiation of PostGrad](image)

Figure 2: Objects are layered entities

A profile may be associated with one of more of the following:

1. A layer of an object;
2. Specific methods of an object; and/or
3. Specific instance variables of an object.

In order to activate a method:

1. Profile(s) protecting all object layers surrounding the method must allow access;
2. The method's specific profile must allow access; and
3. None of the instance variables accessed by the method must deny access.

Typically, a profile will be associated with the object reflecting the object's overall 'sensitivity'. Profiles will be associated with the methods to implement role-based security: the set of operations any user can perform on an object does not only depend on that user's clearance level, but also on the user's function or role in the organization. Profiles which are associated with instance variables will mainly be used as an additional safeguard to ensure that no method accesses information which the subject should not have access to.

An object, message or variable may have more than one profile associated with it, in which case all the profiles must allow access before access will be granted for a message.

In figure 3 Object1 is protected by both Profile1 and Profile2, while Method1 is additionally protected by Profile3 and Profile4. In order to activate Method1, permission will have to be obtained from all four profiles; if any one denies access, Method1 will not be activated.
4 Gates

4.1 Operation of gates

A gate is a mechanism associated with every object, which will intercept any message leaving the object or arriving at the object. Arriving messages will only be allowed to activate the called method if permission can be obtained from the relevant profile(s). In the case of leaving messages, the profiles associated with the sending object will be tagged to the message. Whenever an instance variable with a profile is accessed, the message will be tagged with that profile. Messages spawned by another message will be tagged with all the profiles associated with the spawning message. If a spawned message returns a value, the spawning message will be tagged with the profiles carried by the returning message; if no values are returned the spawning message will resume execution with all the profiles it had when it spawned the returning message.

All messages crossing the borders between object layers will be checked by the gate, verifying baggage when travelling inwards and attaching profiles when travelling outwards. In order to reference any method or variable at an inner layer, all border crossings between the origin of the reference and the referenced variable or method have to be checked. This ensures that information does not flow to a lower classified class by means of inheritance.

4.2 Examples of message tagging

Assume we are working with a payroll system. All employee salaries are protected by a profile P. The payroll object starts sending messages to all employee objects to determine their current salaries. When these messages return the result (the salary) the profile P is attached to the returning message. The payroll object inserts the salaries into its internal table—this implies that a variable is being modified, causing the profile P to be attached to this variable. Although the payroll object now contains the sensitive salaries, it is not possible to retrieve them illegitimately from this object because they are still protected by their original profile P. If this payroll object now sends a message to the cashbook object with this table as a parameter, the profile P will again be sent along and (eventually) be attached to the cashbook’s variable(s) so that it will still not be possible to obtain the sensitive salaries from the cashbook without the (original) authorization.

Another example of tagging profiles to messages is given in figure 4. There are three objects, O1, O2 and O3, six messages, M1 to M6, and eleven profiles, P1 to P11.

If we firstly assume that all messages return values and that the message M1 is sent to object O1, the sequence of events will be as follows:
Figure 4: Tagging profiles to messages
Object | Active message | Profiles tagged to message | Remark |
--- | --- | --- | --- |
O1 | M1 | - | 1 |
O2 | M3 | P1 | 2 |
O3 | M6 | P1,P3,P2 | 3 |
O2 | M3 | P1,P3,P2,P8 | 4 |
O1 | M1 | P1,P3,P2,P8,P3,P2 | 5 |
O2 | M4 | P1,P3,P2,P8,P3,P2,P1 | |
O3 | M5 | P1,P3,P2,P8,P3,P2,P1,P2 | |
O2 | M4 | P1,P3,P2,P8,P3,P2,P1,P2,P9,P8 | |
O1 | M1 | P1,P3,P2,P8,P3,P2,P1,P2,P9,P8,P2 | |

Note the following (numbers refer to the Remark column in the table above):

1. We assume that the message starts without any profiles attached to it.

2. Sending the message M3 to object O2 causes the activity to 'leave' object O1 and be transferred to object O2. The 'leaving' of O1 causes O1's profile P1 to be attached to the message.

3. Object O2 now send the message M6 to O3. This message first 'leaves' the layer of O2 which is protected by P3 and then the layer protected by P2 causing both P3 and P2 to be attached to the message. P1 is still attached to the message because the message was spawned by M3 of O2 which P1 was then associated.

4. Control now returns to method M3 of object O2. Since information is returned (our initial assumption), this information 'leaves' the layer protected by profile P8, causing P8 to be attached to the message.

5. Control has now returned from M3 of O2. Again, since the message returned a value to M1 of O1, 'leaving' from M3 caused profiles P3 and P2 to be attached to the message. (Note that P3 and P2 are already attached to the message—in an optimized environment it is not necessary to attach them again.)

If we assume, on the other hand, that no messages return values, the sequence of events will be as follows if the message M1 is sent to object O1:

Object | Active message | Profiles tagged to message | Remark |
--- | --- | --- | --- |
O1 | M1 | - | 1 |
O2 | M3 | P1 | 2 |
O3 | M6 | P1,P3,P2 | |
O2 | M3 | P1 | 1 |
O1 | M1 | - | 2 |
O2 | M4 | P1 | |
O3 | M5 | P1,P2 | |
O2 | M4 | P1 | 3 |
O1 | M1 | - | 4 |

Note the following (numbers refer to the Remark column in the table above):

1. Control returned to method M3, object O2 from method M6, object O3. Since no value is returned (our assumption) M3 resumes execution with the profiles it had before sending message M6.

2. M1 resumes execution with the profiles it had before sending message M3.

3. M4 resumes execution with the profiles it had before sending message M5.

4. M1 resumes execution with the profiles it had before sending message M4.

4.3 Problems posed by gates

The concept of a gate is the only real deviation in SECDB from the classical object-oriented concept and (not surprisingly) poses the most challenges. These challenges include
1. Handling of messages when access is denied;
2. Cooperation with the object's profile, especially when the profile requires information from the object in order to decide whether access should be granted; and
3. Logical integration into the object-oriented model, especially where such a model is described formally.

Problem (1) can be treated like a normal failure to complete execution of a method, e.g., when the hardware fails or the disk media is unreadable. This will typically involve abortion of the current transaction and rolling the database back. We do not go into further details here.

Polyinstantiation is often used to solve (1): different 'slots' for a single field are maintained for every clearance level; when a process with clearance level \( n \) writes a value to a field, any other process with clearance level \( n \) (or higher) can read that value. However, if a process with clearance level \( m \), where \( m \) is more trusted than \( n \), writes a value to the same field, all processes at clearance level \( m \) and higher will be returned this value when they read the field, while all processes at clearance levels \( n \) to \( m - 1 \) will still be returned the value written by the process at level \( n \). (See [12] for a description of polyinstantiation in object-oriented systems.) Polyinstantiation has to be adapted to be useable in SECDB since data is not necessarily classified at different levels in the PCM model. This means that it is not always possible to determine (and, in any case, it might be dangerous when it is possible) how many 'slots' may exist for the same value. Polyinstantiation will have to be done by attaching 'cascading' profiles to a polyinstantiated object—one level of profile for every copy of the object. If a profile denies access, the profile one level down the cascade is asked for access; this process is continued until a profile allows access; in this case access will be granted to the (copy of the) object on the same level as the granting profile. Only if none of the profiles grant access, will the request be denied. We do not discuss cascaded profiles further in this paper.

Problem (2)—cooperation between an object and its profile—can be illustrated as follows: Suppose some user must have access to every non-manager's employee information. The profile for the employee objects must then first determine whether a specific employee is a manager before granting access. If the fact whether an employee is a manager or not forms part of the employee information, this implies that a profile must sometimes access an object which may not be accessed by the sender of the original message. A similar, but slightly more difficult restriction to solve by ad hoc programming, is a rule allowing access to a salary if and only if the salary is less than $100,000. In order to handle this kind of restriction, SECDB allows a profile to access an object (by sending messages to it) without getting permission from a profile. To ensure that profiles are not used to extract protected information from an object, access to the profile may be restricted to come from gates (and possibly from some other highly trusted objects, but not from arbitrary objects).

5 Example

The example in figure 5 is intended to illustrate profiles, profile classes and hierarchies of profiles. In a real application we would like to see a classification scheme based on functions (or roles) in the organization, rather than on traditional classification levels. In figure 5 SECURITY PROFILE is the primitive security profile class. UNCLASSIFIED, RESTRICTED, SYS_ADMIN, etc, are all profile (sub) classes. SECURITY PROFILE, UNCLASSIFIED, RESTRICTED, CONFIDENTIAL, SECRET and TOP_SECRET form a non-decreasing sequence of profile classes. SYS_ADMIN also offers at least as much protection as SECURITY PROFILE. However, no such relationship exists necessarily between SYS_ADMIN and UNCLASSIFIED, RESTRICTED, etc. In the example this is used where some profile classes are intended to protect information according to the classification level of the information and another class, SYS_ADMIN, is used to protect system structures, such as profile objects.

EMPLOYEE_CLASS describes a class of employee objects. When defining this class, the system security officer decided to link it to the CONF_PROFILE profile. Every instance of EMPLOYEE_CLASS,
Figure 5: Example
such as \texttt{EMP.REC}, will then be associated with \texttt{CONF.PROFILE}.

6 Model characteristics

In SECDB a previously defined and instantiated security profile is specified (or inherited) for every class whenever such a class is defined.

Subclasses of a class will, by default, inherit the profile of the superclass. A security profile may, however, be specified when defining a subclass, overriding the inherited profile. Viewing an object as a layered entity implies that security (both the simple security property and the $^*$-property) [1] will not be compromised, even if the subclass has less restricting access requirements than its superclass. The simple security property requires that subjects should not have access to information they do not have permission to access and the $^*$-property requires, as mentioned earlier, that no subject can write information where another subject without the proper authorization can then read it.

The simple security property and the $^*$-property have to be demonstrated for two cases:

1. Where messages are sent from one object to another (possibly at different classification levels); and
2. Where an object inherits a message or variable from some superclass (again possibly at a different classification level).

The following arguments do this for the respective cases:

1. When messages are sent from one object to another, the gate/profile combination(s) of every object will ensure the simple security property. Since information sent along with any message carries with it the profile of any object it leaves, the $^*$-property is ensured: any access to that object will have to go through the original containing object's profile, implying that a lower "clearance" level will still not be allowed to access it, although it may now be contained in an object with a lower classification level.

2. When a message or variable is inherited from a superclass, the profile specified in that superclass will be associated with the corresponding layer of the object. Information contained may therefore not be accessed without permission from that security profile, ensuring the simple security property. Also, any information leaving that layer will be tagged with that layer's profile, ensuring the $^*$-property.

7 PCM in the SECDB environment

As we indicated at the start, SECDB has been designed to use PCM. The primary advantage of PCM is its ability to enforce security in potentially non-secure environments, especially in network environments where the identity of the originator of a request is often concealed by layers of application software, network servers and system software. SECDB also shows promise in a distributed database where multiple system security officers may be involved. (The original description of PCM uses the term 'object' to refer to a static entity and the term 'subject' to refer to a dynamic entity; we use the term 'static entity' in this section to avoid confusion in the object-oriented context.)

PCM uses three concepts:

1. A baggage collector to collect 'baggage' about all software and hardware involved in handling a request; baggage is defined as 'the minimum amount of information that has to be collected and must accompany the access request on its route in order that responsibility and access authority checking can be performed even though various transformations or domain crossings may occur';

2. A profile for a static entity which specifies which items are required to take part in handling a request and/or which items are not allowed to take part when accessing this entity; and
3. A validator to compare collected baggage to the accessed static entity’s profile.

In SECDDB, to fit the object-oriented model, the profile data (containing the allowing and
prohibiting contexts) and the validator have been merged into a single profile object.
Although PCM may be used to protect non-object-oriented databases, the object-oriented model
is more appropriate, because

- Of the similarity between messages in the two models;
- Of the ability to store data, methods and security information as one coherent unit; and
- In the case of the object-oriented model it is possible to consider security implications at any
  abstraction level—specifically it is possible to implement role-based security; in the case of
  other models one can usually only make decisions on whether a subject is allowed to read
  and/or write specific data records or fields.

In an object-oriented database messages, can originate from objects inside the database or from
external sources. Although not the primary goal, SECDDB can be used to ensure that requests
follow an approved route through the database: some objects are for internal use only, or for use
by specific other objects; it is simple to add restrictions to SECDDB profiles to ensure that a request
only comes via an acceptable predecessor helping to ensure the integrity of the database.

8 Implementation considerations

We do not address the implementation considerations in detail in this paper. However, it must be
admitted that a straightforward implementation of the model could be inefficient. Here we point
out four areas where optimization shows promise.

Firstly, an object protected by multiple profiles requires extensive checking. If some of the
associated profiles have stricter requirements than other associated profiles, the less strict profiles
may be ignored. The ordering imposed on profile subclasses stated that a subclass has stricter
access requirements than its superclass(es). This means that if a profile from somewhere low in
the hierarchy has granted access, the profiles from higher in the hierarchy (instances of ancestor
classes) need not be asked for their permission.

Secondly, checks that messages sent inside the environment conform to the profile restrictions
can often be done statically (at compile time or system configuration time).

Thirdly, the need for a profile to send a message to its superclass’ GrantAccess can also be
eliminated in some cases: if it is possible to check statically that the access requirements set by a
profile class are indeed as least as strict as the requirements set by its superclass.

Lastly, it is unnecessary to tag a profile to a message if the specific profile (or an instance of a
descendant class) is already tagged to the message.

It is also worth pointing out that the association of a profile with an object is not expensive: this
association can be done by having a pointer (often from the class and not the individual objects) to
the relevant profile. This will not require much more memory than a simple sensitivity level would
have taken.

9 Comparison with other models

Very few models have been proposed for secure object-oriented databases.

One promising model for a secure object-oriented database system was proposed by Keefe, Tsai
and Thuraisingham [7]. Their model, known as SODA, assigns a classification (sensitivity) level
to a protected object. Every message which travels through the system, carries with it a current
classification level and a clearance level. The current classification level is adjusted whenever an
object with a higher classification is accessed. Rules, based on the current classification level and the
clearance level and on an object's classification level, determine whether access should be granted to an object.

It can be shown that SODA is similar to a special case of SECDB: define baggage to consist of
1. The clearance level of the sender of a message; and
2. The sensitivity level of any information accessed
and define profiles to contain the sensitivity levels of objects they protect. Note however, that the primary concern in SODA is the protection of variables and objects, whereas methods are the primary concern in SECDB, with variables and objects secondary.

SORION [12] is another multilevel secure object-oriented data model. This model assigns security levels to subjects and entities. A set of rules based on these security levels restricts access.

The access control language of Mizuno and Oldehoeft [10] is based on extended access control lists (ACLs): a four-tuple ACL entry
1. Specifies which user may activate a method;
2. Through which class the request may come;
3. Through which specific object (class instance) the request may come; and
4. Lastly names the protected method.

SECDB is more general; however note that SECDB does not consider classes encountered on the route of the request, but rather objects—ie instances of classes.

Two other methods which may be used to ensure that a request follows an acceptable internal route through the database have been suggested in the literature: views [6] and a law-based approach [9]. In the case of views, an object may define different views—or interfaces—for different objects. A profile in SECDB can specify which objects may access the protected object and also which methods are available to those objects, similar to the view concept. However, this specification in SECDB is not limited to the immediate predecessor object involved in a request. In the law-based approach a set of Prolog rules or laws may be specified to manage the exchange of messages in the system. This is a very general (and powerful) mechanism, enabling (and aimed at) the specification of basic properties, such as inheritance.

10 Further research

Research still remains to be done in the following areas
1. Automating profile generation and automated validation of profiles (for consistency);
2. Identification of unnecessary baggage in order to keep baggage as small as possible;
3. Implications of multiple inheritance on the model; and
4. Designing a notation for specifying security constraints (especially in real-world systems).
5. The model should be applicable to distributed systems where requests may come from many sites, often not even directly connected to the database site. However, the distributed systems may operate concurrently, and the effects of this on the model should be investigated.

11 Conclusion

Object-oriented systems is based on the premise that objects are self-contained entities consisting of data and code. It is meaningful to expect that objects should also have the responsibility to protect themselves. Further, protection in object-oriented systems must fit as cleanly as possible into the generally accepted object-oriented paradigm. In this paper we have proposed a model, SECDB, for supporting security in an object-oriented database, which satisfies these criteria.
SECDB is based on the Path Context Model (PCM), which means that the entire access path is taken into account when it is decided whether a request should be allowed to proceed. This information collected while traversing the access path, is known as baggage. Two concepts are added to the object-oriented paradigm: profiles and gates. Profiles contain criteria for restricting access to an associated object. Gates are mechanisms which will compare baggage accompanying a request to the relevant profile(s), and then either allow or disallow the request to proceed. Profiles are objects themselves. They are also designed to be inherited by a subclass similar to the way instance variables are inherited.

Although SECDB is based on PCM, properties of traditional security models such as the Bell and LaPadula model can still be supported. We illustrated this by indicating how SECDB can emulate SODA, a model for a secure object-oriented database based on the Bell and LaPadula model.

References


Modelling the Algebra of Weakest Preconditions

Abstract: Dijkstra's weakest precondition semantics, as presented in textbook form by Gries, may be viewed as an equational algebra. The problem then is to find a reasonable (set-theoretic) model of this algebra. This paper provides one.

Keywords: Guarded commands, Invariants, Predicate Transformers, Weakest preconditions.


1 Introduction

Programming language semantics comes in three flavours: operational, denotational and axiomatic. Of the latter, one of the best known is Dijkstra's weakest precondition semantics, introduced in Dijkstra [4] and [5], presented in textbook form in Gries [12], offered as a programming methodology in Backhouse [2], Dromey [7] and Morgan [15], and rejuvenated by Dijkstra and Scholten [6] as well as Holt [13].

The idea is that the meaning of a (nondeterministic) program $\alpha$ is entirely given by describing, for any given predicate $Q$ which is desired to be true upon termination of $\alpha$, the set of all those states $s$ such that execution of $\alpha$ from $s$ will result in $\alpha$ terminating in a state where $Q$ is true. This set is called the weakest precondition of $\alpha$ with respect to $Q$, written $wp(\alpha,Q)$. By a state we mean the usual notion of a sequence of values of all program variables. The set of all states is here denoted by $S$. A predicate is in the first instance an interpreted formula in a first-order language. However, it is common practise to equate a predicate $Q$ with the set of all those states in which $Q$ is true, and we adopt this useful ambiguity without further mention. Predicates therefore are subsets of $S$. A program is nondeterministic if from a given input state different output states are possible. This holds in particular for atomic programs, so that if a program is viewed (operationally) as a sequence of atomic steps then we may think of an execution tree of states developing from any initial state.

Note that for any program $\alpha$, $\langle wp(\alpha,\cdot) \rangle$ is a mapping from predicates to predicates. Such a mapping is called a predicate transformer from the power set $P(S)$ (i.e. the set of all subsets of $S$) into itself. The Gries/Dijkstra methodology is to characterise programs by axiomatising the behaviour of predicate transformers. Since the axiomatisation is equational it is appropriate to think of it as an algebra, and it is this Gries/Dijkstra algebra to which our title refers.
The algebra of weakest preconditions has matured considerably from the original five 'healthiness conditions' in Dijkstra [5] to the self-contained algebraico-logical presentation of Dijkstra and Scholten [6]. For example, Dijkstra and Scholten ([6] p125) embrace a notion for which Dijkstra ([5] p76,77) could see no use: that of unbounded nondeterminism. (That is, from any given state there may be infinitely many possible next states for any program \( \alpha \).) We choose as representative the presentation of Gries [12], which, for the benefit of the reader, we quickly outline.

We will denote programs by \( \alpha, \beta, \gamma, \ldots \), and predicates by \( P, Q, R, \ldots \) (This notation differs from Gries.) There are certain atomic programs called \( \text{skip}, \text{abort} \) and (in Dijkstra and Scholten [6]) \( \text{havoc} \). Another atomic program is the assignment statement \( 'z := e' \), where \( z \) is a program variable and \( e \) is some expression (e.g. arithmetical). From the atomic programs compound programs are constructed in one of three ways. First, any two programs \( \alpha \) and \( \beta \) may be composed into another program \( \alpha;\beta \), intuitively read as 'do \( \alpha \), then do \( \beta \). Second, for any predicates \( B_1, B_2, \ldots, B_n \) and programs \( \alpha_1, \alpha_2, \ldots, \alpha_n \) there is an alternative command IF of the form:

\[
\text{if } B_1 \land \alpha_1 \land B_2 \land \alpha_2 \land \ldots \land B_n \land \alpha_n \text{ fi,}
\]

intuitively read as 'select some true \( B_i \) and execute the corresponding \( \alpha_i \). [Note: the \( B_i \)'s are called guards — which is why the Gries/Dijkstra language is sometimes referred to as a guarded command language.] Third, there is an iterative command DO of the form:

\[
\text{do } B_1 \land \alpha_1 \land B_2 \land \alpha_2 \land \ldots \land B_n \land \alpha_n \text{ od,}
\]

intuitively read as 'Repeat the following until no longer possible: select some true \( B_i \) and execute the corresponding \( \alpha_i \).

The Gries/Dijkstra aim is to try and capture these intuitive meanings by conditions imposed on the 'wp(\( \alpha, - \)') predicate transformer. In Gries [12] these are the following: (We use the Gries numbering as an aid to the reader.)

**The Algebra of Weakest Preconditions**

(7.3) Law of the Excluded Miracle: \( \text{wp}(\alpha, \emptyset) = \emptyset \)

(7.4) Distributivity of Conjunction: \( \text{wp}(\alpha, Q) \land \text{wp}(\alpha, R) = \text{wp}(\alpha, Q \land R) \)

(7.5) Law of Monotonicity: If \( Q \subseteq R \) then \( \text{wp}(\alpha, Q) \subseteq \text{wp}(\alpha, R) \)

(7.6) Distributivity of Disjunction: \( \text{wp}(\alpha, Q) \cup \text{wp}(\alpha, R) \subseteq \text{wp}(\alpha, Q \cup R) \)

(7.7) For any deterministic program \( \alpha \), \( \text{wp}(\alpha, Q) \cup \text{wp}(\alpha, R) = \text{wp}(\alpha, Q \cup R) \)
\[(8.1) \quad \text{wp}(\text{skip}, Q) = Q\]
\[(8.2) \quad \text{wp}(\text{abort}, Q) = \emptyset\]
\[(8.3) \quad \text{wp}(\alpha; \beta, Q) = \text{wp}(\alpha, \text{wp}(\beta, Q))\]
\[(9.1.3) \quad \text{wp}(z := e, Q) = \{ s \mid s[e/z] \in Q \}\]
\[(10.3b) \quad \text{wp}(\text{IF}, Q) = (B_1 \cup B_2) \cap (\neg B_1 \cup \text{wp}(\alpha_1, Q)) \cap (\neg B_2 \cup \text{wp}(\alpha_2, Q))\]
\[(11.2) \quad \text{wp}(\text{DO}, Q) = \bigcup_{n \geq 0} H_n(Q), \text{ where } H_0(Q) = \neg B \cap Q, \text{ and } H_{n+1}(Q) = H_n(Q) \cup \text{wp}(\text{IF}, H_n(Q))\]

The reader familiar with Gries [12] will note that since our aim is to consider principles rather than details we have effected some simplifications, which we now explain. First, we have omitted multiple assignment (Gries [12] 9.4), since it merely repeats single assignment. Second, we have restricted IF to at most two guards (i.e. 'if \( B_1 \rightarrow \alpha_1 \) \[ B_2 \rightarrow \alpha_2 \) fi') since it is a mere notational matter to extend the results obtained for two guards to \( n \) guards. Third, we have restricted DO to one guard (i.e. 'do \( B \rightarrow \alpha\) od', or more familiarly 'while \( B \) do \( \alpha\)') since, as both Gries ([12] p139) and Dijkstra and Scholten ([6] p188) point out, in the presence of the general IF command the simple DO will suffice. Fourth, the notation \( s[e/z] \) is explained in §3.

As algebras go, one would expect the algebra of weakest preconditions to have some perspicuous standard model, preferably a set-theoretic one. (As, for example, a standard model for Boolean algebras is the calculus of sets.) But the matter is not that simple. The intuitive model of the algebra of weakest preconditions in Gries [12], for example, is an opportunistic hybrid of first-order logic, set theory and operational semantics, and it is not clear how this could give rise to a set-theoretic model. It seems that such a model must be reasonably sophisticated, in the sense of not just considering the input-output relation. Such an input-output approach has been explored for example in Sanderson [16], Blikle [3] and Schmidt and Ströhlein [18]. Programs are modelled as binary (input-output) relations over \( S \), and operations from the calculus of relations model operations on programs. This model has some pleasant features, but as is clearly pointed out in Gordon [9], it falters on an important point: (8.3) is not satisfied when composition of programs is interpreted as composition of relations.

Our aim in this paper is to model programs as next-state relations. On this approach a program run yields an execution sequence of states; because programs are nondeterministic any initial state \( s \) gives rise to an execution tree, and the meaning of a program is given by the set of all its possible execution sequences. Such a set will be called a flowset (by analogy
with flow diagram). In §2 we develop a calculus of flowsets, and this is used in §3 to verify all
the Gries/Dijkstra conditions listed above. Thus the calculus of flowsets models the algebra
of weakest preconditions. In §4 we briefly consider invariants (in an algebraic setting) in order

We also have a secondary aim, which is pedagogical. Increasingly, Mathematics students
are also doing Computer Science, and vice versa. To a mathematically mature student pop-
ulation weakest precondition semantics can quickly and neatly be explained in the algebraic
context of predicate transformers. But then the algebra should have a standard model.

2 The Calculus of Flowsets

Let $S$ be any set, the elements of which are called states. We will model computations by se-
quencies of states, called 'execution sequences', or exseqs for short. Terminating computations
are modelled by finite sequences, non-terminating ones by infinite sequences. Amongst termi-
nating computations we distinguish those which terminate cleanly from those which don't.
In the latter case we say the computation aborts, and we model this by having the exseq for
that computation ending in a special symbol '$\perp$', with $\perp \notin S$.

We define the set $\text{Seq}(S)$ of exseqs as follows:

1. $S^+$ denotes the set of all finite non-empty sequences of elements of $S$.
2. $S^\perp$ denotes the set of all finite nonempty sequences with $\perp$ as the last component
   and elements of $S$ as the first and (if any) intermediate components.
3. $S^\infty$ denotes the set of all infinite sequences of elements of $S$.
4. $\text{Seq}(S) = S^+ \cup S^\perp \cup S^\infty$.

Note that $S^+, S^\perp$ and $S^\infty$ are disjoint. These three sets naturally represent (respec-
tively) cleanly terminating computations, computations terminating in abortion, and non-
terminating computations. We denote exseqs by $x$, $y$, $z$, ... etc, the idea being that when
we write `$x = (x_1, x_2, x_3, ...)$' it is left open whether or not $x$ is finite. But when we write
`$x = (x_1, x_2, ..., x_n)$' then $x$ is finite and has $x_n$ as last element, where either $x_n \in S$ (if
$x \in S^+$) or $x_n = \perp$ (if $x \in S^\perp$).

We now define some operations on exseqs. For any $x \in \text{Seq}(S)$, say $x = (x_1, x_2, x_3, ...)$:

1. $\text{first}(x) = x_1$
\[
\text{length}(x) = \begin{cases} 
n & \text{if } x \in S^+ \cup S^\perp \text{ and } x = (z_1, z_2, \ldots, z_n) \\
\infty & \text{if } x \in S^\infty
\end{cases}
\]

\[
\text{last}(x) = \begin{cases} 
x_n & \text{if } x \in S^+ \cup S^\perp \text{ and } x = (z_1, z_2, \ldots, z_n) \\
\text{undefined} & \text{if } x \in S^\infty
\end{cases}
\]

We also need to be able to \textit{compose} two exseqs \(x\) and \(y\). In the paradigm case this takes place when \(x\) terminates in exactly that state in which \(y\) begins: then \(x \circ y\) is obtained by identifying \(x\)'s last component with \(y\)'s first component, thus joining the two exseqs together. For other cases special provision must be made. As follows:

\begin{itemize}
\item[(3)] For any \(x, y \in \text{Seq}(S)\),
\[
\begin{aligned}
x \circ y &= \begin{cases} 
x & \text{if } x \in S^\perp \cup S^\infty \\
(x_1, x_2, \ldots, x_n, y_2, y_3, \ldots) & \text{if } x \in S^+, \text{ say } x = (x_1, x_2, \ldots, x_n) \text{ and } y = (y_1, y_2, y_3, \ldots), \text{ and } \text{last}(x) = \text{first}(y) \\
\text{undefined} & \text{otherwise}
\end{cases}
\end{aligned}
\]
\end{itemize}

The idea is that if a computation aborts, or does not terminate, then formally sequencing another computation after it has no effect. Sequencing is only effective when the second computation picks up where the first left off.

Finally, we make explicit the usual \textit{prefix ordering} of sequences — also called ‘ordering by initial subsequences’. For any \(x, y \in \text{Seq}(S)\), we have (with \(n < \infty\) for every \(n \in \mathbb{N}\)):

\begin{itemize}
\item[(4)] \(x \leq y\) \iff (i) \(\text{length}(x) \leq \text{length}(y)\), and \(\text{length}(x) = \text{length}(y)\)
\item[(ii)] \(z_i = y_i\) for every \(i\) such that \(0 \leq i \leq \text{length}(x)\).
\end{itemize}

We can now explore some mathematical properties of our little calculus of exseqs. To begin with, \(\circ\) is associative and \(\leq\) is a partial order. Thus:

\begin{itemize}
\item[(5)] **Lemma** \((\text{Seq}(S), \circ, \leq)\) is a partially ordered semigroup.
\end{itemize}

As is often the case in dealing with partially ordered structures (e.g. in denotational semantics), it will be important that chains have least upper bounds. Recall that a \textit{chain} \(C\) in a partially ordered set \((X, \leq)\) is a linearly ordered subset of \(X\) — i.e. one such that for any \(x, y \in C\) either \(x \leq y\) or \(y \leq x\). We denote least upper bounds by the neutral notation ‘lub’.

\begin{itemize}
\item[(6)] **Lemma** Every chain in \(\text{Seq}(S)\) has a least upper bound.
\end{itemize}

\textbf{Proof} Let \(\{x_i\}_{i \in I}\) be a chain of exseqs (under the prefix ordering). Define \(x\) to be that exseq such that (i) \(\text{length}(x) = \text{lub}\{\text{length}(y) \mid y \in \{x_i\}_{i \in I}\}\) and (ii) \(\forall y \in \{x_i\}_{i \in I}\) and \(\forall j : 0 \leq j \leq \text{length}(y)\) we have \(y_j = x_j\). Intuitively, each \(x_i\) is a prefix of \(x_{i+1}\), and \(x\)
is that exseq of which all the \( x_i \)'s are prefixes. Also, \( \text{lub}(\mathcal{N}) = \infty \). We leave it as an exercise to show that \( \mathbf{x} = \text{lub} \{ x_i \}_{i \in I} \). \( \square \)

Exseqs will model computations, but our aim is more ambitious than that: we wish to model the programs from which these computations arise. We do so by using the \textit{power construction} expounded (e.g.) in Goldblatt [8] and Grätzer and Whitney [17]. That is, from the partially ordered semigroup \((\text{Seq}(\mathcal{S}), \circ, \leq)\) we can form its \textit{power structure} \((\mathcal{P}(\text{Seq}(\mathcal{S})), \circ, =, \mathbb{E})\), where:

(7) \( \mathcal{P}(\text{Seq}(\mathcal{S})) \) is the set of all subsets \( X, Y, Z, \ldots \) of \( \text{Seq}(\mathcal{S}) \);

\[
X \circ Y = \{ x \circ y \mid x \in X \text{ and } y \in Y \} \quad \forall X, Y \in \mathcal{P}(\text{Seq}(\mathcal{S}));
\]

\( X \rightarrow Y \) iff \((\forall x \in X)(\exists y \in Y)[x \leq y] \) and \((\forall y \in Y)(\exists x \in X)[x \leq y] \)

\( \forall X, Y \in \mathcal{P}(\text{Seq}(\mathcal{S})); \)

\( \mathbb{E} = \{(s)|s \in S \} \) (i.e. the set of one-component sequences).

[Note: Readers familiar with powerdomain theory in denotational semantics will recognise \( \circ \) as the \textit{Egli-Milner ordering}.]

The required model of programs is obtained as a substructure of this power structure, namely that consisting of certain special sets of exseqs.

(8) A set \( X \in \mathcal{P}(\text{Seq}(\mathcal{S})) \) of exseqs is called a \textit{flowset} if it satisfies

Axiom 1: \( \forall s \in S \exists x \in X \) with \( \text{first}(x) = s \), and

Axiom 2: \( \forall x, y \in X, x \neq y. \)

The set of all flowsets will be denoted by \( \mathcal{F} \).

The idea with Axiom 1 is that any state \( s \) is a possible initial state of any program. A computation may not actually progress from \( s \) (in which case it is modelled by the exseq \( (s) \)), or it may immediately abort (modelled by the exseq \( (s, \bot) \)) — but at any rate it is \textit{defined}. One virtue of this idea is that, for any state \( s \), any set of exseqs \( \mathbf{x} \) with \( \text{first}(\mathbf{x}) = s \) provides a natural model for the \textit{execution tree} (or ‘extree’ for short) of some program \( \alpha \) from this initial state \( s \). Note that it is risky to rely on a graphical presentation of such trees, or a definition intended to capture such a graphical presentation. For example, a program \( \alpha \) may, from some given initial state \( s \), have the possible computation sequences \((s, t_1, u, v_1)\) and \((s, t_2, u, v_2)\), and only these. Yet the graphical tree representation of Figure 1 would seem to indicate that \((s, t_1, u, v_2)\) and \((s, t_2, u, v_1)\) are also possible execution sequences. But this is not intended.

The idea with Axiom 2 is that no initial subsequence of an execution sequence is also an

247
execution sequence. This, in fact, is the manifestation of a rather subtle point concerning nondeterminism. Generally speaking, a program is said to be nondeterministic if from any given state there is no assumption of a unique next state. The question raised here, which does not seem to have been addressed before, is whether or not one should uniformly assume the existence of a next state. In short: does the notion of nondeterminism allow a program which has already proceeded up to some state s to sometimes terminate at s and sometimes not? Space does not allow a discussion of this issue here, so we resolve the matter by simply choosing one of the alternatives: Axiom 2 rules out a notion of nondeterminism on which initial subsequences of computations can also be computations. Note that any extree satisfies Axiom 2 (but not necessarily Axiom 1).

The operation \( \circ \) between flowsets is intended to model sequential composition. Think of \( X \) and \( Y \) respectively as the set of all possible computations of programs \( \alpha \) and \( \beta \). Then \( X \circ Y \) corresponds to the set of all possible computations of the program which consists of doing \( \beta \) immediately after \( \alpha \). Technically: for any sequence \( x \in X \), if it does not terminate cleanly leave it; if it does then append at its final state (say) \( x_n \) all exseqs in \( Y \) starting with \( x_n \). The relation \( \equiv \) between flowsets is the power order of the prefix relation between exseqs. It says that \( X \equiv Y \) iff any exseq in \( X \) is a prefix of some exseq in \( Y \), and any exseq in \( Y \) has some exseq in \( X \) as prefix. Thus, intuitively, \( Y \) extends \( X \). The set \( E \) is useful for technical reasons: it will model a program called \( \text{null} \), which from any initial state \( s \) does exactly nothing.

We now come to the calculus of flowsets.

(9) **Theorem** \( (F, \circ, E, \equiv) \) is a partially ordered monoid, with \( E \) as identity for \( \circ \), and as minimum under \( \equiv \).

**Proof** To establish the monoid part of the Theorem we need to check that the power operation \( \circ \) is associative, and that \( E \) is a (left- and right-) identity for \( \circ \). The former follows since the power operation of any associative operation is again associative. The
latter is also easy, but it is not immediate. That $X \circ E = X$ and $E \circ X = X$ for any flowset $X$ must be checked for all possible cases.

Secondly we must establish that $\equiv$ is a partial order, and $E$ its minimum. It is known that the power order of any partial order is a quasi-order — that is, reflexive and transitive. So we only need to check that $\equiv$ is anti-symmetric. To do so, let $X$ and $Y$ be flowsets such that $X \equiv Y$ and $Y \equiv X$; we need to show that $X = Y$. For this we show that $X \subseteq Y$, analogous reasoning would then also establish that $Y \subseteq X$. So let $x \in X$ arbitrarily. Since $X \equiv Y$ there is some $y \in Y$ such that $x \leq y$. But also, since $Y \equiv X$, there is then some $z \in X$ such that $y \leq z$. But then $x \leq z$, hence by Axiom 2 $x = z$, hence since $x \leq y \leq z$ we have $x = y$, hence $x \in Y$. Thus $X \subseteq Y$, as required. To see that $E$ is the minimum element of $\equiv$ we just need to invoke Axiom 1.

The next question to address is whether chains in $\mathcal{F}$ have least upper bounds. Note that the ‘obvious’ response that flowsets are sets and that therefore lub’s should be unions is fallacious. The union of flowsets is indeed a set, but it is not necessarily a flowset: Axiom 2 is easily violated. However, our earlier preparation in Lemma (6) pays off: chains of flowsets will have lub’s because chains of eSEQs have them.

(10) **Theorem** Let $\{A_i\}_{i \in I}$ be any chain of flowsets under the ordering $\equiv$. Let

$$A = \{x \mid x = \text{lub under } \leq \text{ of some chain } C \text{ in } \bigcup_{i \in I} A_i\}.$$ 

Then $A = \text{lub } \{A_i\}_{i \in I}$ under $\equiv$.

In §3 we will view the iterative command DO as a repetition of IF commands. We therefore need to build into the calculus of flowsets the notion of repetition; it is precisely for this purpose that we talked above about chains and lub’s. We define *iterated composition* in the usual way:

(11) For any flowset $X$: 

$$X^0 = E$$

$$X^{n+1} = X^n \circ X, \quad \forall n \geq 0.$$ 

Then:

$$\equiv$$

(12) **Theorem** For any flowset $X$, $(X^n)_{n \geq 0}$ forms a chain under the ordering $\equiv$.

**Proof** We must show that $X^0 \equiv X^1 \equiv X^2 \equiv \ldots$. By Theorem (9) $E \equiv X$ for any flowset $X$, so we only need to show that, $\forall n \geq 0$, $X^n \equiv X^{n+1} = X^n \circ X$. Since composition simply appends eSEQs in $X$ to eSEQs in $X^n$ this is easy to show by consideration of cases.

It now follows from Theorem (10) that for any flowset $X$ the chain $(X^n)_{n \geq 0}$ must have a
lub. This is an important notion, for which we reserve both a notation and a name.

(13) For any flowset $X$, the iteration $X^*$ of $X$ is defined by $X^* = \text{lub}\{X^n\}_{n \geq 0}$.

Finally we introduce into the calculus of flowsets an operator which does not explicitly feature in the Gries/Dijkstra algebra of weakest preconditions, but which is quite useful as an aid in modelling such operations. It is the nondeterministic choice operator which for programs $\alpha$ and $\beta$ would correspond to a program $\alpha \lor \beta$, interpreted as: 'Nondeterministically choose either $\alpha$ or $\beta$ and then run the chosen program'. In the relational semantics briefly discussed in §2 nondeterministic choice is modelled by set-theoretic union. But in the calculus of flowsets this won't do, since the union of two flowsets need not be a flowset. Instead we choose to model $\alpha \lor \beta$ by what is known as Hilbert's epsilon operator.

This operator, $\epsilon$, features strongly in Higher-Order Logic — see for example [1], [9], and [10]. It is a variable-binding operator, like quantifiers, and can be used as a primitive logical symbol. For our purposes, however, it will suffice to make clear its semantics. Namely, the $\epsilon$-operator acts as a choice function: given any set $A$, $\epsilon$ picks out some unknown but fixed element of $A$, which is then denoted by $\epsilon A$. In particular,

(14) For any indexed set $\{X_i\}_{i \in I}$ of flowsets

\[ \epsilon \{X_i\}_{i \in I} \]

denotes some particular unspecified but fixed $X_i$, $i \in I$.

Note that if the indexed set $I$ is finite then $\epsilon \{X_1, X_2, \ldots, X_n\}$ is some particular one of $n$ flowsets. But we make no finiteness constraints on $I$ in (14), in order to cater for the unbounded nondeterminism of Dijkstra and Scholten [6].

In conclusion we point out some related work. Blikle [3] also models programs as sets of computations and presents an algebra of such sets. But Blikle adopts a notion of 'generalised composition', whereas our approach uses the power construction. Another relevant paper is Hoare [14], which models programs as sets of 'possible traces', along the lines of operational semantics. It is interesting that Hoare ([14] p425) proves exactly what we called Axioms 1 and 2 for flowsets.

3 Verifying the Gries/Dijkstra Conditions

Our aim in this section should be clear, and is easy to state. We model each of the programs in the Gries/Dijkstra language by a flowset, each operation on programs by an operation
on flowsets, and we prove the given formulae of the algebra of weakest preconditions in the calculus of flowsets. We also add a few extra features to the Gries/Dijkstra algebra, as an aid to the exposition. (No substantive changes are made).

We adopt the square bracket notation \('[\ ]'\) of denotational semantics to map each program \(\alpha\) onto its meaning \(\llbracket\alpha\rrbracket\), which will be a flowset. We may also specialise this to 'the meaning of a program \(\alpha\) at some initial state \(s\)', written \(\llbracket\alpha\rrbracket(s)\), which is an extree (or execution tree) in the sense of §2: the set of all those exeqs \(x\) in \(\llbracket\alpha\rrbracket\) such that \(\text{first}(x) = s\).

To begin with:

1. \(\llbracket\text{skip}\rrbracket = \{(s, s) | s \in S\}\)
2. \(\llbracket\text{abort}\rrbracket = \{(s, \bot) | s \in S\}\)
3. \(\llbracket\text{havoc}\rrbracket = \{(s, t) | s \in S \text{ and } t \in S\}\)
4. \(\llbracket\text{null}\rrbracket = \{(s) | s \in S\} = E\)

We introduce the atomic program \(\text{null}\) because it will be useful in defining IF. Next, we model sequential composition of programs by composition of flowsets. That is:

2. For any programs \(\alpha\) and \(\beta\), \(\llbracket\alpha; \beta\rrbracket = \llbracket\alpha\rrbracket \circ \llbracket\beta\rrbracket\).

The assignment statement, as is well known, is problematic in the Gries/Dijkstra algebra insofar as it is the only command which deals directly with program variables. But the problems raised by assignment are extraneous to the modelling proposed here, and does not affect the issues we discuss. We therefore simply adopt the Gries/Dijkstra technique (also commonplace in other contexts) of indicating notationally a change in state effected by a change in the value of a program variable. Namely:

3. For any state \(s \in S\), \('s[e/z]'\) will denote that state which differs from \(s\) only in that the value of the program variable \(z\) is replaced by the value of the expression \(e\) evaluated in \(s\).

It is then easy to model the assignment statement as a flowset. [Note: Like Gries ([12] p118), we assume \(e\) to be well-defined in every \(S\).]

4. For any program variable \(z\) and expression \(e\),

\[
\llbracket z := e \rrbracket = \{(s, s[e/z]) | s \in S\}
\]

To model IF we use both the Hilbert epsilon operator and the null command. We first define nondeterministic choice:

5. For any programs \(\alpha\) and \(\beta\), \(\llbracket\alpha \lor \beta\rrbracket = \epsilon(\llbracket\alpha\rrbracket, \llbracket\beta\rrbracket)\).
We then define the meaning of IF at an arbitrary initial state $s$, thus obtaining an extree, and take the union of all of these extrees. As follows:

(6) For any programs $\alpha_1$ and $\alpha_2$, any predicates $B_1$ and $B_2$, and any state $s \in S$, the meaning of \textquote{\(if \ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2 \ fi\)} at $s$ is given by:

\[
[if \ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2 \ fi](s) = \begin{cases} 
[\alpha](s) & \text{if } s \in B_1 \text{ and } s \notin B_2 \\
[\beta](s) & \text{if } s \notin B_1 \text{ and } s \in B_2 \\
[\alpha \lor \beta](s) & \text{if } s \in B_1 \text{ and } s \in B_2 \\
[null](s) & \text{if } s \notin B_1 \text{ and } s \notin B_2 
\end{cases}
\]

and the meaning of IF itself is:

\[
[if \ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2 \ fi] = \bigcup_{s \in S} [if \ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2 \ fi](s).
\]

[Note: We use, as is customary in this context, \(s \in B\) as an abbreviation for \(B\) is true in state $s$. Likewise \(s \notin B\) means \(B\) is false in $s$.]. The intention here should be clear. How does \(if \ B_1 \rightarrow \alpha_1 \ [] \ B_2 \rightarrow \alpha_2 \ fi\) execute from any initial state $s$? If $B_1$ is true but $B_2$ is not $\alpha_1$ will be executed from $s$; if $B_1$ is false but $B_2$ is true then $\alpha_2$ will be executed; if both are true exactly one of $\alpha_1$ and $\alpha_2$ will be nondeterministically selected and executed, and if neither $B_1$ nor $B_2$ is true nothing will happen. Two typical special cases of IF are \(if \ B \rightarrow \alpha\), which we equate to \(if \ B \rightarrow \alpha \ [] \neg B \rightarrow \text{null } fi\), and \(if \ B \rightarrow \alpha \ else \beta \ fi\), which we equate to \(if \ B \rightarrow \alpha \ [] \neg B \rightarrow \beta\). It is then easy to read off their meanings from (6), above.

With Gries and Dijkstra we assume the guards to be well-defined in every state, so that the four possibilities enumerated above are the only ones. On the other hand, unlike Gries ([12] p132), Dijkstra ([5] p34) and Dijkstra and Scholten ([6] p144) we do not say that if no guard is true then IF aborts. There is danger of confusion here, since in fact in Dijkstra ([5] p26) and Dijkstra and Scholten ([6] p135) 'abort' really means 'does not terminate', so what they are actually saying is that if no guard is true then IF goes into an endless and unproductive loop. It is not clear to us what the virtues are of such a notion. What will be made clear by our exposition, we trust, is that having (literally) the \textit{null}-option available makes for a tidy treatment of IF, both technically and conceptually. On our treatment, in any composition 'IF;\alpha' control will always pass from IF to $\alpha$, even when no guard of IF is true. The reader is reminded that our aim is to model the \textit{algebra of weakest preconditions}, and that in doing so we are not constrained by any particular intuitive \textit{semantics} of the constructs involved. A final point: each extree by assumption satisfies Axiom 2 (of §2), hence a union of extrees.
over every state \( s \in S \) will be a flowset. Thus \([[ if \; B_1 \rightarrow \alpha_1 \; ] \; \; B_2 \rightarrow \alpha_2 \; fi]]\) is well-defined as a flowset.

For the iterative command \( \text{DO} \), in the simple form 'while \( B \) do \( \alpha \)', our earlier preparation really pays off. The definition is quite simple: \( \text{DO} \) is the iteration of \( \text{IF} \).

\[ [[ \text{while } B \text{ do } \alpha ]] = [[ if \; B \text{ do } \alpha ]^n] = \text{lub}\{[[ if \; B \text{ do } \alpha ]^n]_{n \geq 0}\} = \text{lub}\{[[ if \; B \text{ do } \alpha ]]_{n \geq 0}\} \]

Recalling the relevant definitions in §2, we see that \( \text{DO} \) is defined as the least upper bound of the chain of flowsets arising from repeating the \( \text{IF} \) command. [Note: This is the second equality in (7). The third is easy to prove by induction.] Intuitively, to perform 'while \( B \) do \( \alpha \)' consists of repeatedly doing the following, until it has no further effect: check whether \( B \) is true, and if so do \( \alpha \).

We now come to the central definition, which is that of weakest precondition: for any program \( \alpha \) and predicate \( Q \), Dijkstra ([5] p16,17), Gries ([12] 7.1 p108) and Dijkstra and Scholten ([6] p129) are unanimous that 'wp(\( \alpha, Q \))' must denote the set of all those states such that execution of \( \alpha \) begun in one of them is guaranteed to terminate, and when it does it satisfies \( Q \). Our definition will capture this intuition, but it adds a clarification: 'wp(\( \alpha, Q \))' will denote the set of all states from which \( \alpha \) terminates cleanly (and satisfies \( Q \) upon termination). In our context, if \( \alpha \) does not terminate cleanly (i.e. aborts) it cannot satisfy \( Q \), since \( \bot \) is not a state.

\[ \text{wp}(\alpha, Q) = \{ s \in S | (\forall x \in [[ \alpha ]]_s)(x \in S^+ \text{ and } \text{last}(x) \in Q)\} \]

We can now verify the formulae of §1.

(9) **Theorem** For any program \( \alpha \) and predicates \( Q \) and \( R \):

(a) (Gries [12], 7.3) Law of the Excluded Miracle: \( \text{wp}(\alpha, \emptyset) = \emptyset. \)

(b) (Gries [12], 7.4) Distributivity of Conjunction:

\[ \text{wp}(\alpha, Q) \cap \text{wp}(\alpha, R) = \text{wp}(\alpha, Q \cap R). \]

(c) (Gries [12], 7.5) Law of Monotonicity:

\[ \text{if } Q \subseteq R \text{ then } \text{wp}(\alpha, Q) \subseteq \text{wp}(\alpha, R) .\]

(d) (Gries [12], 7.6) Distributivity of Disjunction:

\[ \text{wp}(\alpha, Q) \cup \text{wp}(\alpha, R) \subseteq \text{wp}(\alpha, Q \cup R). \]

**Proof** All of these depend upon simple logical properties, such as in (b) the distribution of universal quantification over conjunction.
To check that in our context the converse of Theorem (9)(d) does not hold in general an example such as that of Gries ([12] p111) would suffice. But the converse does hold for deterministic programs.

(10) A program $\alpha$ is said to be deterministic iff for every $s \in S \llbracket \alpha \rrbracket (s)$ is a singleton set.

That is, from any initial state $\alpha$ can proceed to execute in exactly one way.

(11) **Theorem** For any predicates $Q$ and $R$, and any deterministic program $\alpha$


\[(Gries \ [12] \ 7.7) \ \ wp(\alpha, Q) \cup wp(\alpha, R) = wp(\alpha, Q \cup R).\]

The atomic programs are easy to characterise from Definition (1).

(12) **Theorem** For any predicate $Q$

\[(a) \ (Gries \ [12], \ 8.1) \ \ wp(skip, Q) = Q.\]
\[(b) \ (Gries \ [12], \ 8.2) \ \ wp(abort, Q) = \emptyset.\]
\[(c) \ (Dijkstra \ and \ Scholten \ [6], \ 7.12) \ \ wp(havoc, Q) = S\]
\[(d) \ \ wp(null, Q) = Q.\]

We now come to composition: the place where the relational model fails.

(13) **Theorem** For any programs $\alpha$ and $\beta$, and any predicate $Q$,

\[wp(\alpha; \beta, Q) = wp(\alpha, wp(\beta, Q)).\]

**Proof** Left to right: Let $s \in wp(\alpha; \beta, Q)$, then by (8) any $z \in \llbracket \alpha; \beta \rrbracket (s)$ terminates cleanly, and in $Q$. To show that $s \in wp(\alpha, wp(\beta, Q))$, let $u \in \llbracket \alpha \rrbracket (s)$ arbitrarily. If $u \in S^\perp$ or $u \in S^\omega$ then also by (2.3) (and (2)) we would have $u \in \llbracket \alpha; \beta \rrbracket (s)$, which would then contradict the fact that $u$ must terminate cleanly. So $u \in S^+$, hence $last(u) \in S$. To show that $last(u) \in wp(\beta, Q)$, consider any $v \in \llbracket \beta \rrbracket (last(u))$. Then $u \circ v \in \llbracket \alpha; \beta \rrbracket (s)$, hence by assumption $u \circ v$ terminates cleanly, and in $Q$. But then $v$ must terminate cleanly, and in $Q$. Hence $last(u) \in wp(\beta, Q)$, as required. Analogous reasoning establishes the right to left inclusion. \(\Box\)

As with the definition of the assignment statement we also pass lightly over its weakest precondition result: issues such as definability and non-classical conjunction raised by Gries ([12] 9.1.1) are not germane to our discussion. What we should do is check:

(14) **Theorem** For any program variable $z$, expression $e$ and predicate $Q$:

\[wp(\text{'z := e'}, Q) = \{s \mid s[e/z] \in Q\}.\]

**Proof** Let $s \in wp(\text{'z := e'}, Q)$ then every $x \in \llbracket z := e \rrbracket (s)$ terminates cleanly and in $Q$. But by definition (4) $x = (s, s[e/z])$; hence $s[e/z] \in Q$. For the reverse direction, let $s$ be
such that \( s[e/z] \in Q \). To show that \( s \in wp(z := e', Q) \) take any \( x \in \llbracket z := e \rrbracket(s) \), then by definition (4) \( x = (s, s[e/z]) \). Hence \( x \) terminates cleanly and in \( Q \); so \( s \in wp(z := e', Q) \).

Since we use the (nondeterministic) choice operator in our modelling of IF we determine also the weakest precondition for \( V \). It is quite neat:

(15) **Theorem** For any programs \( \alpha \) and \( \beta \), and any predicate \( Q \),

\[
wp(\alpha \lor \beta, Q) = wp(\alpha, Q) \cap wp(\beta, Q).
\]

**Proof** Left to right: Let \( s \in wp(\alpha \lor \beta, Q) \). Then \( \forall x \in \llbracket \alpha \lor \beta \rrbracket(s) \) we must have that \( x \) terminates cleanly and in \( Q \). But \( \llbracket \alpha \lor \beta \rrbracket(s) \) can be either \( \llbracket \alpha \rrbracket(s) \) or \( \llbracket \beta \rrbracket(s) \), which means that every \( x \) in \( \llbracket \alpha \rrbracket(s) \) must terminate cleanly and in \( Q \), and so must every \( x \) in \( \llbracket \beta \rrbracket(s) \).

Hence \( x \in wp(\alpha, Q) \cap wp(\beta, Q) \). Right to left: Similar.

For the IF operator we are required to verify Gries ([12] 10.3). Aiming in this direction, we get:

(16) **Theorem** For any programs \( \alpha \) and \( \beta \), and any predicates \( B_1, B_2 \) and \( Q \),

\[
wp(\text{if } B_1 \rightarrow \alpha_1 \mid B_2 \rightarrow \alpha_2 \text{ fi}, Q) = (B_1 \cup B_2) \cap (\neg B_1 \cup wp(\alpha_1, Q)) \cap (\neg B_2 \cup wp(\alpha_2, Q)) \cup [\neg B_1 \cap \neg B_2 \cap Q].
\]

[Note: \( \neg B \) denotes 'not B', or (set-theoretically) the complement of B].

**Proof** Left to right: Let \( s \in wp(\text{if } B_1 \rightarrow \alpha_1 \mid B_2 \rightarrow \alpha_2 \text{ fi}, Q) \). Then every \( x \in \llbracket \text{if } B_1 \rightarrow \alpha_1 \mid B_2 \rightarrow \alpha_2 \text{ fi} \rrbracket(s) \) terminates cleanly and in \( Q \). To show \( s \) is in the right hand side we consider two cases: either \( s \in B_1 \cup B_2 \) or \( s \in \neg(B_1 \cup B_2) \). The latter is easy for then by definition (6) \( \llbracket \text{if } B_1 \rightarrow \alpha_1 \mid B_2 \rightarrow \alpha_2 \text{ fi} \rrbracket(s) = \llbracket \text{null} \rrbracket(s) \); hence \( s \in \neg B_1 \cap \neg B_2 \cap Q \). In the former case either \( s \in B_1 \) or \( s \notin B_1 \). If \( s \in B_1 \) then \( \llbracket \text{if } B_1 \rightarrow \alpha_1 \mid B_2 \rightarrow \alpha_2 \text{ fi} \rrbracket(s) = \llbracket \alpha_1 \rrbracket(s) \); hence \( s \in wp(\alpha_1, Q) \subseteq \neg B_1 \cup wp(\alpha_1, Q). \)

On the other hand if \( s \notin B_1 \) then evidently \( s \in \neg B_1 \cup wp(\alpha_1, Q) \) thus \( s \in \neg B_1 \cup wp(\alpha_1, Q) \) in any case. Similarly, \( s \in \neg B_2 \cap wp(\alpha_2, Q) \). Hence \( s \in [B_1 \cup B_2] \cap \neg B_1 \cup wp(\alpha_1, Q) \) \( \cap \neg B_2 \cup wp(\alpha_2, Q) \). Right to left: By a similar consideration of all possible cases.

To read this theorem colloquially, consider this: either some guard in IF is true, or no guard is. In the former case \( wp(\text{IF}, Q) \) is an intersection of three facts: some guard is true, if \( B_1 \) is true then we have \( wp(\alpha_1, Q) \), and if \( B_2 \) is true then we have \( wp(\alpha_2, Q) \). This is an exact transcript of Gries ([12] 10.3). But as pointed out earlier, our treatment differs from that of Gries/Dijkstra also covering explicitly the case where no guard is true: in that case

255
wp(IF, Q) is just Q. It seems to us that this better captures the intuition behind IF than the Gries/Dijkstra idea that IF is non-terminating when no guard is true.

But there is yet a better way of expressing wp(IF, Q):

(17) **Theorem** For any programs α and β, and any predicates B₁, B₂, and Q,

\[
wp(if\ B₁ → α₁ \ ||\ B₂ → α₂ \ fi, Q) = [B₁ \cap \neg B₂ \cap wp(α₁, Q)] \cup [\neg B₁ \cap B₂ \cap wp(α₂, Q)] \cup [B₁ \cap B₂ \cap wp(α₁ \lor α₂, Q)]
\]

\[
\cup [\neg B₁ \cap \neg B₂ \cap Q].
\]

**Proof** Left to right: Let \( s \in wp(if\ B₁ → α₁ \ ||\ B₂ → α₂ \ fi, Q) \). Then every \( x \in \llbracket if\ B₁ → α₁ \ ||\ B₂ → α₂ \ fi\llbracket(s) \) terminates cleanly and in Q. To show \( s \) is in the right hand side we distinguish four mutually exclusive and jointly exhaustive cases:

either \( s \in B₁ \cap \neg B₂ \) or \( s \in \neg B₁ \cap B₂ \) or \( s \in B₁ \cap B₂ \) or \( s \in \neg B₁ \cap \neg B₂ \). We only consider the first case; the others are similar: If \( s \in B₁ \cap \neg B₂ \) then by definition (6) \( \llbracket if\ B₁ → α₁ \ ||\ B₂ → α₂ \ fi\llbracket(s) = \llbracket α₁\llbracket(s) \); hence every \( x \in \llbracket α\llbracket(s) \) terminates cleanly and in Q. So \( s \in B₁ \cap \neg B₂ \cap wp(α₁, Q) \). Right to left: Similar. \( \square \)

This presentation of wp(IF, Q) exactly parallels the definition of IF in (6). That is, wp(IF, Q) breaks down as follows: If the first guard is true and the second false we are dealing with wp(α₁, Q); if the first guard is false and the second true we are dealing with wp(α₂, Q); if both guards are true we are dealing with the weakest precondition of one of α₁ or α₂ (without knowing which), and if no guard is true the weakest precondition is Q itself. In thus explicitly presenting all possible cases the formulation is superior to that of (16). [Note: It is a nontrivial exercise to show directly that the two formulations are equivalent.]

For ‘if B do α’ and ‘if B do α else β fi’ we get as special cases from (17):

(18) **Corollary** For any programs α and β, and any predicates B and Q,

\[
wp(if\ B \ do \ α, Q) = [B \cap wp(α, Q)] \cup [\neg B \cap Q]
\]

\[
wp(if\ B \ do \ α \ else \ β \ fi, Q) = [B \cap wp(α, Q)] \cup [\neg B \cap wp(β, Q)]
\]

As Gries ([12] p135) points out we are often not interested in the weakest precondition of IF, but only in determining if a known precondition implies it. Gries’s Theorem (10.5) gives necessary and sufficient conditions under which \( X \subseteq wp(IF, Q) \) is true. The diligent reader will have no problem with proving this theorem in our context.

Finally, we come to the iterative command, ‘while B do α’. Like Gries ([12] p140) we define for any given predicate Q a sequence \( H_n(Q) \) of predicates, where \( H_n(Q) \) represents the set of all states from which execution of DO terminates (cleanly!) in \( n \) or fewer iterations,
with $Q$ true. But our definition simplifies that of Gries. Namely:

(19) For 'while $B$ do $\alpha$' define predicates $H_n(Q)$, $n \geq 0$, by:

$$H_0(Q) = \neg B \cap Q$$

$$H_{n+1}(Q) = wp(if \ B \ do \ \alpha, H_n(Q)), \ \forall n \geq 0.$$  

The simplification is possible because of our treatment of IF in the case where no guards are true. To prove that nothing is omitted by the simplification we require two Lemmas.

(20) Lemma $\neg B \cap H_n(Q) = \neg B \cap Q, \ \forall n \geq 0$.

Proof By induction.  

This says that any state from which DO terminates in $n$ or fewer iterations, and in which $B$ is false, is also a state in which $Q$ is true.

(21) Lemma $H_{n+1}(Q) = [\neg B \cap Q] \cup [B \cap wp(\alpha, H_n(Q))], \ \forall n \geq 0$.

Proof Use (19),(18) and (20).

This gives exactly the form of Gries's $H_{n+1}(Q)$. An inductive argument then suffices to show that it is also the same set. So we get:

(22) Theorem $H_n(Q)$ as defined by Gries ([12] p140) on the basis of his definition (10.3b, p132) of $wp(IF,Q)$ is the same set $\forall n$ as $H_n(Q)$ defined in definition (19) on the basis of $wp(IF,Q)$ given in Theorem (16) (or (17)), arising from our definition (6) of IF.

It remains to verify Gries ([12] 11.2).

(23) Lemma For 'while $B$ do $\alpha$', and any predicate $Q$, $\{H_n(Q)\}_{n \geq 0}$ forms a chain under the set-theoretic ordering $\subseteq$.

This means that Gries really characterises $wp(DO, \alpha)$ as the least upper bound of the chain (under $\subseteq$) of the $H_n(Q)$'s. And so do we. Again we need two Lemmas. Both demonstrate our approach of defining DO in terms of IF.

(24) Lemma $H_n(Q) = wp((if \ B \ do \ \alpha)^n, \neg B \cap Q), \ \forall n \geq 0$.

Proof By induction. [Note: For any program $\alpha$, $\alpha^0 = null$ and $\alpha^{n+1} = \alpha^n; \alpha$.]

(25) Lemma For any $s \in S$, if $[while \ B \ do \ \alpha](s)$ is finite then $\exists n \in N$ such that $[while \ B \ do \ \alpha](s) = [(if \ B \ do \ \alpha)^n](s)$, and $t \not\in B$ for every leaf $t$ of this extree.

Proof By saying 'the tree is finite' we mean 'only has branches of finite length'. (Since we are dealing with unbounded nondeterminism there may well be infinitely many branches).

257
From (7) we get that \([\text{while } B \text{ do } \alpha] = \text{ lub}\{[(\text{if } B \text{ do } \alpha)^n]\}_{n \geq 0}\), hence if for any particular \(s \in S \ [\text{while } B \text{ do } \alpha](s)\) is finite it follows from Theorem (2.10) that there must be a least number \(m \in \mathcal{N}\) such that
\[\left(\text{if } B \text{ do } \alpha\right)^m(s) = \left(\text{if } B \text{ do } \alpha\right)^{m+1}(s) = \ldots\]
But then \([\text{while } B \text{ do } \alpha](s) = \left(\text{if } B \text{ do } \alpha\right)^m(s)\), and \(t \not\in B\) (since otherwise \([\text{if } B \text{ do } \alpha](s)\) would extend \([\text{if } B \text{ do } \alpha](s)\)).

The idea here is quite simple: a terminating DO from an initial state \(s\) is precisely the \(n\)-fold composition of an IF-statement, for some \(n \in \mathcal{N}\).

(26) **Theorem** (Gries [12] 11.2) For any program \(\alpha\) and predicates \(B\) and \(Q\),
\[\text{wp}(\text{while } B \text{ do } \alpha, Q) = \bigcup_{n \geq 0} H_n(Q) = \bigcup_{n \geq 0} \text{wp}(\text{if } B \text{ do } \alpha)^n, \neg B \cap Q).\]

**Proof** Left to right: Let \(s \in \text{wp}(\text{while } B \text{ do } \alpha, Q)\). Then every exseq \(x\) in \([\text{while } B \text{ do } \alpha](s)\) terminates cleanly and in \(Q\). Hence \([\text{while } B \text{ do } \alpha](s)\) is finite, so by Lemma (25) \(\exists m \in \mathcal{N}\) such that \([\text{while } B \text{ do } \alpha](s) = \left(\text{if } B \text{ do } \alpha\right)^m(s)\) (and \(t \not\in B\) for every leaf \(t\) of this extree). But then every exseq \(x \in \left(\text{if } B \text{ do } \alpha\right)^m(s)\) terminates cleanly and in \(Q\), hence in \(\neg B \cap Q\). Thus by Lemmas (24) and (23), \(s \in \text{wp}(\text{if } B \text{ do } \alpha)^m, \neg B \cap Q) = H_m(Q) \subseteq \bigcup_{n \geq 0} H_n(Q)\).

Right to left: Let \(s \in \bigcup_{n \geq 0} \text{wp}(\text{if } B \text{ do } \alpha)^n, \neg B \cap Q)\), say \(s \in \text{wp}(\text{if } B \text{ do } \alpha)^m, \neg B \cap Q)\) for some \(m \in \mathcal{N}\). Then every exseq \(x \in \left(\text{if } B \text{ do } \alpha\right)^m(s)\) terminates cleanly and in \(\neg B \cap Q\). But then
\[\left(\text{if } B \text{ do } \alpha\right)^m(s) = \left(\text{if } B \text{ do } \alpha\right)^{m+1}(s) = \ldots\]
and hence \([\text{if } B \text{ do } \alpha]^m(s) = (\text{lub}\{[(\text{if } B \text{ do } \alpha)^n]\}_{n \geq 0})(s) = [\text{while } B \text{ do } \alpha](s)\) by (7). Hence every \(x\) in \([\text{while } B \text{ do } \alpha](s)\) terminates cleanly and in \(Q\) and so \(s \in \text{wp}(\text{while } B \text{ do } \alpha, Q)\).

4 **Invariants**

With DO, as with IF, the weakest precondition is not always the most useful precondition. As Gries ([12] p140) puts it:

The formal definition of DO is not easy to use, and gives no insight into developing programs. Therefore, we want to develop a theorem that allows us to work with a useful precondition of a loop (with respect to a postcondition) that is not the weakest precondition.

258
This sought-after precondition is to be called an invariant of the loop: it is '... a predicate \( P \) that is true before and after each iteration of [the loop]' (Gries [12] p141). So the idea is that if \( s \in P \) and DO is executed from \( s \), then the final state is again an element of \( P \). To model the notion of an invariant in our context we come forward with two suggestions.

(1) **Suggestion 1** Instead of restricting the notion of an invariant to loops, define it for any program \( \alpha \).

That is, for any program \( \alpha \) a predicate \( I \subseteq S \) will be called an invariant of \( \alpha \) iff \( \forall s \in I \), if \( \alpha \) is executed from \( s \) then every final state is again an element of \( I \). The reader will no doubt see the immediate problem: \( \alpha \) may not terminate cleanly, or may not terminate at all, so that an appropriate final state may not exist. For this we have:

(2) **Suggestion 2** Think of invariants by analogy with subalgebras: 'I is an invariant of \( \alpha \)' is analogous to a subset \( \alpha \) of an algebra being closed under a given operation.

The virtue of this suggestion is that the problem just raised has been exhaustively investigated in Universal Algebra, and so we may borrow from there. Since \( \alpha \) may not terminate (cleanly, or at all), we may think of it as analogous to a partial operation in an algebra. The question of how to define invariants for programs which do not terminate cleanly is then analogous to this: *What is the correct notion of subalgebra for partial algebras?* For this, consider the comment of Grätzer ([11] p79):

For algebras there is only one reasonable way to define the concepts of subalgebra, homomorphism and congruence relation. For partial algebras we will define three different types of subalgebra, three types of homomorphism, and two types of congruence relation ... all of these concepts have their merits and their drawbacks, and each particular situation determines which one should be used.

Space constraints disallow further discussion, so we simply report that of the three kinds of subalgebra considered by Grätzer we may use two in the present context to give the following alternative notions of an invariant of a program \( \alpha \).

(3) **Alternative 1** A predicate \( I \) is called an invariant of a program \( \alpha \) iff \( \forall s \in I \) the extree \( [\alpha](s) \) is finite and all leaves are \( \in I \).

(4) **Alternative 2** A predicate \( I \) is called an invariant of a program \( \alpha \) iff \( \forall s \in I \), if the extree \( [\alpha](s) \) is finite then all its leaves are \( \in I \).
In the Gries/Dijkstra formulation of invariants termination is not built in — it must be proved separately by a bound function. Thus Gries/Dijkstra implicitly select Alternative 2, hence, for current purposes, so do we.

Having defined the notion of invariant, why is it useful? The idea is that an invariant, as a precondition of a loop, is easier to obtain than the weakest precondition. Namely, for ‘while B do α’ and a given postcondition Q, if we can find an invariant I such that \( \neg B \cap I \subseteq Q \), then I will be a precondition of the loop — that is, \( I \subseteq \wp(\text{while } B \text{ do } \alpha, Q) \).

The reasoning is that if execution is started in I it remains in I; upon termination B is false (hence \( \neg B \) is true). But then any final state is in \( \neg B \cap I \), hence in Q. This crops up in Dijkstra ([5] p38) as 'The Basic Theorem for the Repetitive Construct' (also 'The Fundamental Invariance Theorem for Loops'), in Gries ([12] p144) as Theorem (11.6) ('a theorem concerning a loop, an invariant and a bound function') and in Dijkstra and Scholten ([6] p180) as the 'Main Repetition Theorem'.

Our version is:

\[ (5) \quad \textbf{Theorem} \quad \text{For any predicates } I, B \text{ and } Q, \text{ and any program } \alpha, \text{ if} \]

\[ (a) \quad \neg B \cap I \subseteq Q, \text{ and} \]

\[ (b) \quad B \cap I \subseteq \wp(\alpha, I), \text{ and} \]

\[ (c) \quad \llbracket \text{while } B \text{ do } \alpha \rrbracket(s) \text{ is finite.} \quad \forall s \in I, \]

\[ \text{then} \]

\[ I \subseteq \wp(\text{while } B \text{ do } \alpha, Q). \]

\[ \textbf{Proof} \quad \text{It suffices to show that } I \subseteq \wp(\text{while } B \text{ do } \alpha, I) \text{ since by (a) and monotonicity of } \wp, \wp(\text{while } B \text{ do } \alpha, \neg B \cap I) \subseteq \wp(\text{while } B \text{ do } \alpha, Q) \text{ and using an inductive argument it is easy to verify that } \wp(\text{while } B \text{ do } \alpha, I) = \wp(\text{while } B \text{ do } \alpha, \neg B \cap I). \]

So let \( s \in I \) then either \( s \in B \) or \( s \in \neg B \). If \( s \in \neg B \) then \( \llbracket \text{while } B \text{ do } \alpha \rrbracket(s) = \llbracket (if \ B \ do \ \alpha)^0 \rrbracket(s) = \text{null}(s) = \{(s)\} \). Hence every exseq in this extree terminates cleanly and in I, so \( s \in \wp(\text{while } B \text{ do } \alpha, I) \). Now suppose \( s \in B \). We must show that \( \forall x \in \llbracket \text{while } B \text{ do } \alpha \rrbracket(s), \ x \text{ terminates cleanly and in } I \). By (c) and Lemma (3.25) \( \exists m \in N \) such that \( \llbracket \text{while } B \text{ do } \alpha \rrbracket(s) = \llbracket (if \ B \ do \ \alpha)^m \rrbracket(s) \) (and \( t \notin B \) for every leaf of this extree). So we need only show that every exseq \( x \) in \( \llbracket (if \ B \ do \ \alpha)^m \rrbracket(s) \) terminates cleanly and in I. But \( \llbracket (if \ B \ do \ \alpha)^m \rrbracket = \llbracket if \ B \ do \ \alpha \rrbracket^m \), hence any such \( x \) has \( m \) nested initial subsequences \( x_i, 1 \leq i \leq m \), such that \( \text{last}(x_i) \in B \) for \( 1 \leq i < m \). But then since \( s = \text{first}(x) \in B \cap I \) we get from (b) that \( \text{last}(x_i) \in I \) for \( 1 \leq i \leq m \). Hence, in particular, \( \text{last}(x) \in I \), as required. \[ \square \]
References


MODEL
PROCESS
p1: 0..2;
p2: 0..2
VAR
x : 0..1
TRANS
(0, *, *) → (1, 0, 0);
(1, *, 1) → (1, 0,-1);
(2, *, *) → (-2, 0, 1);
(*, 0, *) → (0, 1, 0);
(*, 1, 1) → (0, 1,-1);
(*, 2, *) → (0,-2, 1)
START
(0,0,1)
SPEC
AG((p1 = 1) => AF(p1 = 2))
END.

Figure 1: A transition system representing the mutual exclusion problem

Figure 2: Reachability graph of a transition system
3.1 Basic Concepts

The language of classical propositional logic is adopted wholesale. As is customary in modal-type logics it is next assumed that there is a set $S$ of states such that every atomic proposition is or is not true at a particular state. The states are related to each other by an accessibility relation. In modal logics the set of states, together with the accessibility relation and the assignment of truth values to atomic propositions in every state, is called a Kripke structure. Some properties are usually ascribed to the accessibility relation; depending on what these are, different modal logics arise. Here the accessibility relation structures the set of states into a tree, the branches representing all possible execution sequences from some initial state. The truth value of a compound formula at a given state may depend on the truth values of some of its subformulae at other states further down the tree. To express this it is necessary to quantify over the states in any particular execution sequence, and also over execution sequences. For the first purpose the notation of modal logic is used—"□" for "always", "◊" for "sometimes" and "□" for "next". For the second purpose the standard notation of first-order logic is used—"∀" for "for all" and "∃" for "there is". Finally, these different quantifiers are combined to obtain six different modalities: ∀□, ∀◊, ∃□, ∃◊ and ∃◊. Thus "∀□α" would say that formula α is true for all states in every execution sequence, "∀◊α" says that in every execution sequence there is some state in which α is true, "∀α" says that α is true in every immediate successor state, and so on.

In the more technical section 3.2 below an until operator is introduced to increase the expressiveness of the language. Roughly, "α U β" is the claim that β will be true at some point in the future, and up to the immediately preceding point α will be true. Again, this may be preceded by ∀ or ∃, indicating respectively that the claim holds for all or only some execution sequences.

3.2 The Branching Time Logic CTL

The alphabet of CTL comprises: a set of variables $P = \{p_1, p_2, \ldots, p_n, \ldots\}$; the constants true and false; the connectives ¬ and ∧; the temporal operators ∀, ∃, ◊ and U with parentheses and square brackets as punctuation symbols.

Any variable by itself is a formula. If α and β are formulae, then so are:

$$\neg \alpha, \alpha \land \beta, \forall \alpha, \exists \alpha, \forall[\alpha U \beta], \exists[\alpha U \beta],$$

and nothing else is a formula. To define validity our assumptions concerning time are packed into the formal definition of a Kripke structure. It is a triple $K = (S, R, v)$ such that:

- $S$ is a finite set, the elements of which are called states. There is some distinguished state $s_0 \in S$, called the initial state.
• \( R \) is an adjacency relation over \( S \), such that \((S, R)\) is a tree, with root node \( s_0 \).

• \( v : P \times S \rightarrow \{0, 1\} \) is an assignment of some truth value (0 or 1) to every variable at every state.

A path is a branch of the tree \((S, R)\)—an infinite sequence of states \((s_0, s_1, \ldots)\) starting with the root node \( s_0 \) and such that \((\forall i)[(s_i, s_{i+1}) \in R] \). A path from a given state \( s \) is a branch of the subtree with \( s \) as root.

Having assumed an assignment of truth values to every atomic formula at every state an inductive definition can be given of what it means for any compound formula \( \alpha \) to be true at some state \( s \). Such a fact is indicated by \( s \models \alpha \) and the fact that \( \alpha \) is not true at \( s \) by \( s \not\models \alpha \). Inductively then, for any state \( s : s \models \text{true}, \) and \( s \not\models \text{false} ; \) for any atomic formula \( p \), and any state \( s : s \models p \) iff \( v(p, s) = 1 \); for any formulae \( \alpha \) and \( \beta \), and any state \( s \):

• \( s \models \neg \alpha \) iff \( s \not\models \alpha \)

• \( s \models \alpha \land \beta \) iff \( s \models \alpha \) and \( s \models \beta \)

• \( s \models \forall \alpha \) iff for every state \( t \) such that \((s, t) \in R \) we have \( t \models \alpha \)

• \( s \models \exists \alpha \) iff there exists a state \( t \) such that \((s, t) \in R \) and \( t \models \alpha \)

• \( s \models \forall[\alpha \land \beta] \) iff for all paths \((t_0(= s), t_1, t_2, \ldots)\) from \( s \) \((\exists i)[i \geq 0 \) and \( t_i \models \beta \) and \( (\forall j)[0 \leq j < i \) implies \( t_j \models \alpha \)]

• \( s \models \exists[\alpha \land \beta] \) iff there exists a path \((t_0(= s), t_1, t_2, \ldots)\) from \( s \) such that \((\exists i)[i \geq 0 \) and \( t_i \models \beta \) and \( (\forall j)[0 \leq j < i \) implies \( t_j \models \alpha \)]

The other propositional connectives, \( \lor \) ("or"), \( \Rightarrow \) ("implies") and \( \iff \) ("iff") may be defined from \( \neg \) and \( \land \) in the ordinary textbook way. The remaining four of the six modalities mentioned in Section 3.1 can now be introduced by definition:

\( \forall \alpha \iff \forall[\text{true } \alpha] \); \( \exists \alpha \iff \exists[\text{true } \alpha] \); \( \forall \alpha \iff \neg \exists \neg \alpha \) and \( \exists \alpha \iff \neg \forall \neg \alpha \).

### 3.3 Specification

By the inductive definition of \( \models \) every CTL formula \( \alpha \) is or is not true at any state \( s \) in a Kripke structure \( K \). Because of the forward looking nature of the temporal operators it is necessary, for unspecified \( \alpha \), to have full knowledge of the distribution of truth values to atomic formulae at all states in the subtree with root \( s \) in order to deduce the truth value of \( \alpha \) at \( s \). Conversely, of course, if we do know that \( \alpha \) is true at \( s \) we know something about the subtree with root \( s \). The convention is adopted of saying that a subtree with root \( s \) has property \( \alpha \) if the state \( s \) itself has property \( \alpha \), which is to say that \( \alpha \) is true at \( s \). In particular then, a Kripke structure \( K \) has property \( \alpha \) iff \( \alpha \) is true at the root node \( s_0 \).
A combination of transition system and logic is now possible. A reactive system is represented by a transition system from which an execution tree can be computed. Repetitive sequences of states are discarded according to the rules of fairness (see Section 4.3) and thus the tree can be reduced to be finite without losing important information. Simple tests on the variables of the transition system are used to determine the truth value of atomic propositions. The reduced tree may thus be regarded as a Kripke structure, and desirable properties of the reactive system are expressed as CTL formulae. The CTL formula therefore represents a property the system should have while the (finite) execution tree of the reactive system represents a Kripke structure. The model checker can now determine whether the given reactive system has the specified property by checking whether the formula is true at the root node of the tree. Temporal logic can be used to express important properties of reactive systems such as freedom from deadlock, absence of starvation and responsiveness. The CTL formula appearing after the keyword “SPEC” in Figure 1 captures absence of starvation for process 1 of the given transition system. (For practical reasons a slightly different notation is used in the computerised system for CTL—“AG” meaning “∀O” and “AF” meaning “∀◊”). CTL is therefore seen as a query language to formulate questions about the system being studied, as suggested by Everitt[7]. A list of properties which are relevant for reactive systems is given in [10].

4 An Efficient Model Checker

The model checker described here executes the transition system in order to explore various paths while determining the truth value of the CTL formula. The paths explored are determined by the specific CTL formula. For example, the formula ∀Oα will force the model checker to explore all paths leading from the initial state (unless the formula is found to be violated before all paths have been explored), while the formula ∃◊α will allow the model checker to stop as soon as some path is found which leads to a state in which α is true.

Although in theory efficient model checking algorithms exist for some suitably restricted temporal logics such as CTL, little has been reported about the performance of model checker implementations. Even recently published algorithms such as the algorithm given in [12] are inefficient and it was therefore decided to explore the various possibilities of designing a model checker which would be efficient enough to verify real reactive systems such as large protocols. The success of model checking depends on efficient state space exploration techniques. Such techniques have been investigated thoroughly in the field of protocol validation and this experience influenced the design of the model checker described here.

Many protocol validators generate states dynamically, testing each state against a predefined set of correctness criteria known as state properties. To avoid analysing unnecessary states it is necessary to determine whether each newly generated state
had been visited before. It is thus necessary to compare each new state to all previously generated states. Therefore all unique states must be stored and as the number of states increases it takes progressively longer to determine the uniqueness of a state. It was determined experimentally that state comparison is the most time consuming operation in traditional protocol validators[8]. About 100 states per second can be processed on medium scale machines, rendering the technique impractical for systems which generate more than about $10^4$ states.

Traditional model checkers[3, 4, 1, 12] compute a reachability graph which is stored in memory. Computation of this graph leads to a similar efficiency problem: each new state must be compared against all previous states to ensure uniqueness. Although few measurements of the performance of model checkers exist, this similarity between model checkers and protocol validators suggests that systems which generate more than about $10^4$ states cannot be analysed by traditional model checkers. Burch et al.[2] showed that one large problem could be analysed by representing the state space symbolically. However, only the potential size of the state space ($10^{20}$) is given and not the number of actually reachable states. Furthermore, the technique is not generally applicable and more research will be required to determine its usefulness in practice.

Fortunately Holzmann recently found a new method of searching large state spaces[8, 9] which leads to a significant improvement in performance. States are generated dynamically by representing the system by a state vector model. The traditional method of determining uniqueness of states is replaced by a very efficient one. The new technique requires a large vector of bits to be maintained in memory to keep track of previously generated states but, even so, much larger systems can be analysed before space becomes a problem. Holzmann measured the performance of the technique and showed that it can be used to analyse protocols generating up to $10^7$ states.

The model checker described here uses this efficient method to speed up the computation of the reachability graph. In addition, it is unnecessary to store the reachability graph because model checking can be done while generating the reachability graph. This approach has several important advantages. Firstly, space is saved because no reachability graph is stored explicitly. Secondly, the truth value of many temporal formulae can be determined without generating the entire reachability graph. An example of such a formula is $\exists \diamond \alpha$ which will be satisfied as soon as a state is found in which $\alpha$ is true. This makes the model checker faster. Thirdly, problems which generate a reachability graph which is too large for the available memory space could sometimes be analysed because it may be unnecessary to generate the entire graph. To put these ideas into practice two problems had to be solved, namely, how to handle fairness constraints and nested formulae without storing the reachability graph explicitly.
4.1 Fast State Comparison

As explained in Section 2 the state of a transition system is described by its state vector. The state vector as well as the guards and actions are represented by bitstrings of fixed length. The start state is assigned to the state vector to initialise a given transition system. To execute the transition system the guard of each transition is compared to the contents of the state vector until one is found to match. That transition is then selected and a new state is generated by adding the corresponding action to the state vector, and the process is repeated to execute a depth-first search of the state space. Each state is viewed as a (unique) index into a large array of bits (called a bit vector) as illustrated in Figure 3. As each new state is generated the corresponding bit in the bit vector is set to record the fact that such a state has been generated. The bit vector is used to determine whether a newly generated state is unique or not.

Unfortunately it is not enough to keep track of states which have been visited. The current execution path needs to be recorded in order to detect loops. This is important to handle fairness as will be explained in Section 4.3. A stack of state records is therefore kept to record the current execution path. To decide whether a state is on the stack, the entire stack must be searched—a relatively expensive operation. However, a state can only be on the stack if it has been visited before and therefore the bit vector provides a fast way of avoiding a stack search when it is unnecessary. Because loops are detected, the model checking algorithm will always terminate and when a specification is violated a counterexample is given which helps the user to find his error.

4.2 Model Checking Algorithm

The language used for presenting the algorithm is (more or less) standard Modula-2, for which see [13].

For brevity only a description of the various procedures used in the code is given and declarations are left out. With respect to the algorithm of Figure 4, note that

- There are two modes of operation: trans denoting that transitions are being
LOOP
  CASE mode OF
  trans:
    IF (i <= transMax) THEN
    IF Enabled(i) THEN
      stack[depth].index:= i+1;
      s:= Action(i);
      IF StateVisitedBefore THEN
        IF Stacked(s) THEN (* loop detected *)
          mode:= pred
        ELSE (* visited on earlier path *)
          WITH stack[depth] DO
            s:= state; i:= index
          END
        END
      END
    ELSE (* unique state *)
      value:= TruthValue(sf);
      MarkStateAsVisited;
      IF value = T THEN
        UpdateStack(sf); mode:= pred
      ELSIF value = F THEN
        mode:= pred
      END
    END
    ELSE (* next transition *)
    INC(i); mode:= trans
  END
  ELSE (* no more transitions *)
  value:= FinalValue(sf);
  DEC(depth);
  mode:= pred
  END
  pred:
  IF StackEmpty THEN RETURN value = T
  ELSE
    WITH stack[depth] DO
      s:= state; i:= index
    END;
    AdaptControlInfo
  END
END
END

Figure 4: Model Checking Algorithm
tested in order to generate new states and \textit{pred} indicating a backtrack operation to a predecessor state.

- A stack of state records is kept, each stack entry representing a state along the current path. Each record contains two fields: \textit{state}—a state descriptor and \textit{index}—indicating which transition to try next.

- Each state is mapped onto a unique bit in a vector of bits and \textit{State Visited} is a test to determine whether the bit corresponding to the state \textit{s} is set.

- Procedure \textit{Enabled} returns TRUE if the guard of transition \textit{i} evaluates to true in the current state \textit{s}, and FALSE otherwise.

- Procedure \textit{Action} returns a new state which is derived from the current state \textit{s} by adding each component of the action vector of transition \textit{i} to each component of \textit{s}.

- Procedure \textit{Stacked} returns TRUE if state \textit{s} has been visited before along the current path, and FALSE otherwise.

- Procedure \textit{Truth Value} returns the truth value of formula \textit{sf} in the current state, three values being possible: \textit{F} (false), \textit{T} (true) or \textit{U} (undefined).

- For every unique state generated the mode is changed depending on the truth value of \textit{sf}. When no further state exploration is necessary the mode is changed to \textit{pred}. While value is \textit{U} state exploration is allowed to proceed. Procedure \textit{Update Stack} has to do with fairness for which see Section 4.3.

- Procedure \textit{Final Value} determines the final truth value of the formula in the current state once it is known that all paths leading from state \textit{s} have been explored. It depends on \textit{value} and on the formula. For example if \textit{value} is \textit{U} and the formula is 3\$\alpha$, \textit{value} is changed to \textit{F}.

- Whenever a predecessor state is entered \textit{Adapt Control Info} changes the values of \textit{mode}, \textit{value} and \textit{depth} depending on whether more state exploration is necessary or not.

\subsection*{4.3 Fairness}

In the present context fairness means that if a transition is enabled it should eventually be allowed to occur. Fairness concerns the behaviour of the transition system when certain paths are executed repeatedly and non-deterministic choices occur. Consider Figure 2 again. The execution path \((0, 0, 1) \rightarrow (1, 0, 1) \rightarrow (1, 1, 1) \rightarrow (1, 2, 0) \rightarrow (1, 0, 1) \rightarrow (1, 1, 1) \cdots\) is an example of an unfair path. If the transition which leads to state \((1, 1, 1)\) is always chosen at state \((1, 0, 1)\), process 1 will never be able to execute. The system will therefore never reach state \((2, 0, 0)\). Under these
circumstances the given specification will not be satisfied. The model checker must ignore such unfair behaviour because the transition system is meant to be fair.

The various temporal formulae can be classified into two groups according to their behaviour with respect to fairness:

1. $\forall \Box \alpha$, $\exists \Diamond \alpha$ and $\exists (\alpha U \beta)$
2. $\exists \Box \alpha$, $\forall \Diamond \alpha$ and $\forall (\alpha U \beta)$

The formulae in group 1 need no special treatment. To handle the second group, however, fairness must be considered. Consider the formula $\forall \Diamond \alpha$. Suppose the truth value of the given formula must be determined in state $s$. Therefore each path leading from state $s$ must lead to a state in which $\alpha$ holds. If some path leads back to $s$ without reaching a state in which $\alpha$ is true, it is necessary to know whether some other path starting at $s$ can lead to a state in which $\alpha$ is true because, if so, the first path represents an unfair path and can be ignored. On the other hand if a state in which $\alpha$ holds cannot be reached from $s$, the given formula is false in $s$. To keep track of this is simple if the reachability graph is kept in memory. If the reachability graph is generated on the fly fairness is handled by keeping information about fairness on the stack.

4.4 Subproblem Detection

Sometimes the truth value of a temporal formula depends on the truth value of another temporal formula. For example The CTL formula given in Figure 1 specifies absence of starvation for process 1: whenever process 1 is trying to enter its critical section ($p_1 = 1$), it will eventually reach it ($p_1 = 2$). Nested formulae can be handled in different ways. A simple method is to break formulae down into subformulae which are then handled in a bottom-up way but much unnecessary work is normally done that way.

The problem can be solved more efficiently in a top-down fashion by computing truth values only when necessary. Consider the CTL formula given in Figure 1 again. To determine the truth value of this formula in state $s$ the model checker will explore all paths leading from $s$ while checking that in each state along every path the argument to $\forall \Box$ is true. The implication makes it unnecessary to determine the truth value of the nested modality in any state in which $p_1 \neq 1$. If $p_1 = 1$ however, the truth value of $\forall \Diamond (p_1 = 2)$ is needed. Tuominen gives a top-down model checking algorithm[12] but the truth values of some subformulae are still computed unnecessarily. For example, the truth value of the nested modality in the given example will be recomputed in all states while it is often possible to deduce its value from some earlier state. Figure 2 provides an example. The truth value of the subformula $\forall \Diamond (p_1 = 2)$ will be needed in states $(1,0,1)$, $(1,1,1)$ and $(1,2,0)$. However, since the latter two states are reachable from the state $(1,0,1)$,
it is unnecessary to determine the truth value of the nested modality in all three states. If the subformula is true in state \((1, 0, 1)\) it is bound to be true in the other two states. If it is false in state \((1, 0, 1)\) the main formula is invalidated and therefore the other two states can be ignored.

A new technique called subproblem detection is proposed which exploits such contextual information to avoid a significant amount of unnecessary processing. Whenever the truth value of some subformula cannot be determined directly in state \(s\) the subformula and state \(s\) are remembered as a subproblem to be analysed at a later stage. The subformula is assumed to be true and the model checker proceeds. In the given example it was necessary to analyse the same subformula in three different states. For each unique subformula a list of states is kept in which its truth value must be determined. Several simplifications are now possible. For example, as soon as the specification is found to be violated any subproblems which have been generated need not be analysed any further. Furthermore, the set of states \(V_s\) visited in order to compute the truth value of a subformula in state \(s\) is recorded in the bit vector. The truth value of the same subformula in some other state \(s'\) can then often be deduced from its truth value in \(s\) if \(s' \in V_s\). The technique has been implemented and found to improve the speed of the model checker significantly.

Another advantage of the technique of subproblem detection is that it provides a natural way to parallelise the model checker: a main processor can be used to detect subproblems while several “worker” processors are used to solve subproblems. Results are returned to the main processor which keeps track of everything in order to determine the final result. Communication overhead is low since little information needs to be exchanged among processors. To solve a particular subproblem a worker processor needs to know only the start state and the particular subformula. The returned result is simply the truth value of the particular subformula in the start state. As an added bonus, a parallel version of the model checker will be using the memory of several machines as a combined resource. States generated in order to detect a particular subproblem need not be regenerated by the processor which is used to solve the subproblem. Similarly different state spaces are usually generated to solve different subproblems. A parallel version of the model checker based on a number of interconnected workstations is currently being developed.

4.5 Using Paging to Save More Space

Traditional model checkers keep information about each unique state in memory in the form of a state graph. For large state spaces this graph will be too large to fit into memory. However, because state generation is so slow when traditional methods are used—typically about 100 states per second—a space problem is not encountered in practice; problems large enough to cause a space problem cannot even be considered because too much processing time will be required.

Model checkers capable of processing at least several thousand states per second on a modern workstation can be built by using the bit vector technique. It
is thus possible to analyse much larger state spaces in an acceptable amount of time. But larger problems require huge virtual address spaces to accommodate the bitvector—too large to be supported directly by currently available operating systems. Fortunately the bit vector is extremely sparse and clustered[8] and therefore multi-level paging techniques can be used to handle large bit vectors. A virtual address space supported by disk slows down the model checker to an unacceptable speed[8] and therefore pages are allocated dynamically in memory as needed. The technique is used to save memory. It depends on the fact that only a small fragment of the virtual address space is usually needed. Results obtained thus far are encouraging but more experimentation will be required to determine the success of this technique.

5 Conclusion

A transition system is used to model the dynamic properties of a system. Figure 1 shows typical input accepted by the model checker. The execution tree of the transition system is generated dynamically while it is being verified whether the given specification is satisfied. To avoid storing the reachability graph explicitly a new technique was developed to handle fairness. To handle nesting another technique (called subproblem detection) was developed. It has several other advantages related to model checking in general. A paging technique is used to save space in order to handle even larger problems.

A model checker based on the suggested design has been implemented and used to verify several systems, the largest thus far being a model of the X.21 protocol. Systems which generate no more than a few thousand states can be analysed by using a personal computer, but a more powerful machine is necessary to analyse larger systems. The suggested techniques enable us to process about 3000 states per second on a workstation based on the Motorola 88K processor. Little has been reported about the efficiency of other model checkers except that about 100 states per second seems to be the norm using a typical workstation.

6 Acknowledgements

Several people have contributed ideas to the model checker described in this paper. Specific acknowledgements are due to Professor Chris Brink of the University of Cape Town, and the following people from the University of Stellenbosch: Dieter Barnard, Werner Fouché, Pieter Muller and Willem Visser.

References


A New Algorithm for finding an Upper Bound of the Genus of a Graph

D.I. Carson and O.R. Oellermann
Department of Computer Science,
University of Natal (Durban), King George V avenue

Abstract
In this paper we discuss the problem of finding an upper bound on the genus of a graph. This problem has applications to circuit layouts. An electronic circuit may be modelled by a graph. By punching holes into the circuit board, one may be able to lay out the circuit so that no two wires cross. The smallest number of holes that are required for a given graph (that models such a circuit) is called the genus of the graph. The number of holes in the surface equals the genus of the surface. Thus, finding an algorithm which approximates the genus of the graph which models the circuit is important. We present a new algorithm for finding an upper bound on the genus of a graph, which uses a combinatorial data structure called the PQ-tree data structure. Four additional PQ-tree templates are used to extend the basic combinatorial reduction process of the PQ-trees to consider a genus approximation algorithm.

Keywords: PQ-trees, genus, circuit layouts

CR Category: G.2.2, E.1

Introduction
The genus of the sphere is 0. Any surface obtained from the sphere by inserting $n \geq 0$ handles on the sphere is called a *surface of genus* $n$. For example, Figure 1(a) shows a surface of genus 1, while Figure 1(b) shows a surface of genus 3.
The genus $g(G)$ of a 2-connected graph $G$ is the minimum among the genera of the surfaces on which $G$ may be embedded (i.e. drawn so that no two of its edges cross). The genus of a 2-connected graph is always defined (see [2]). In this paper we develop an efficient algorithm for finding an upper bound on the genus of a graph.

The basis for the algorithm relies on the PQ-tree data structure. This data structure was developed by Booth and Lueker [3], and allows for the efficient implementation of combinatorial problems, in particular those relating to graph theory. The most important result using PQ-trees is a linear time (in the number of vertices) modification by Booth and Lueker [3] of an algorithm for testing whether a given 2-connected graph is planar [9]. For further applications of the PQ-tree data structure see [4]. Please see Appendix 1 for an introductory discussion of st-numberings of graphs and PQ-trees.

The main feature provided by the PQ-tree data structure in graph theory algorithms is the flexibility allowed when embedding a graph $G$ in the plane. Initially, we embed a single vertex of $G$. Using the PQ-tree data structure, we may add one vertex at a time to the subgraph $H$ constructed, and all possible embeddings of $H$ constructed thus far are represented by the
PQ-tree. Thus, when it is not possible to add the next vertex to H without allowing edges to cross, then we may conclude that the graph is non-planar, and consider our options. It is at this stage that we propose to insert handles into the corresponding embedding of H, and to allow the algorithm to continue.

A Basis for the Algorithm

We follow the notation of [2]. If a graph G has $|E(G)| = q$, and $|V(G)| = p$, then we say G is a $(p, q)$ graph. The genus of a graph equals the sum of the genera of its blocks (Battle, Harary, Kodama and Youngs [1]). Therefore, it suffices to develop algorithms for finding the genus of a 2-connected graph. While Filotti, Miller and Reif [6] developed an $O(p^{O(\gamma(G))})$ algorithm for finding $\gamma(G)$, their algorithm is clearly not efficient. In fact, determining the genus of an arbitrary 2-connected graph efficiently is believed to be a difficult problem. This belief was strengthened in 1989 when Thomassen [10] showed that the following decision problem is NP-complete: For a given graph G and integer $k \geq 0$, can G be embedded on a surface of genus $k$? This suggests trying to find heuristics that approximate the genus of a 2-connected graph. Thus, we consider in this paper an algorithm for finding an upper bound on the genus of a 2-connected graph. Suppose G is an arbitrary non-planar 2-connected graph. Using the PQ-tree data structure we obtain, at each stage of the reduction algorithm, a description of the partial embedding of G. Figure 2 illustrates an st-numbered non-planar graph G. That G is non-planar is easy to see since a subgraph of G isomorphic to the non-planar Kuratowski graph $K_{3,3}$ is induced by the vertex set $\{1, 3, 4, 5, 6, 7\}$, with $\{1, 4, 6\}$ and $\{3, 5, 7\}$ being the two partite sets.
Now, during the reduction algorithm, the reduction for vertex 6 fails, because both the pertinent leaves are children of a Q-node X, and they have an empty child (the leaf representing the edge from vertex 4 to vertex 7) between them, in the order as they appear as children of the Q-node. However, if we insert a handle with ends in regions $R_1$ and $R_2$, as shown in Figure 3(a), then we obtain an embedding of $G$ on the torus, as shown in Figure 3(b). The ideas of embedding the graph $G$ of Figure 2 on the torus, as shown in Figure 3(b), hold the key to our algorithm.

We now generalise the above idea to any non-planar graph. Observe that all we did in the above example was, given a template matching failure, to insert a handle from the one consecutive sequence of pertinent edges to a final region on whose boundary vertex 6 lies. This technique of inserting a handle during the reduction process is the main idea behind our algorithm.
The Algorithm
Suppose we are reducing with respect to some vertex having st-number i, and suppose that, during the reduction, a node X in our PQ-tree T does not match one of Templates P₁, P₂, ..., P₆, Q₁, Q₂, Q₃. Our goal is to delete maximal subsequences of full leaves from Frontier(X) to allow the reduction process to proceed. At the end of the reduction we only have one maximal subsequence of full leaves in Frontier(T). Given a maximal subsequence of full leaves to delete, we identify the edges represented by the full leaves in a pseudo-vertex. The single remaining subsequence of full leaves in Frontier(T), after the reduction is complete, is identified in a vertex called the base. We place a handle from each pseudo-vertex to the base. Finally we undo the edges identified at each pseudo-vertex and place the corresponding identified edges along that handle at the pseudo-vertex so that they are now identified at the base vertex.

Let T be the PQ-tree equivalent to the maximal subtree rooted at a Q-node X such that in Frontier(X) there are as few maximal subsequences of full leaves as possible. Suppose that there are k maximal subsequences of full
leaves in Frontier(T'). We define Pert\_i(X) to be the sequence of pertinent children of X whose full leaf descendants are precisely the i-th maximal sequence of full leaves in Frontier(T') as we proceed from left to right in Frontier(T').

We define an operation Reduce(Pert\_i(X)) to simplify Pert\_i(X) in the following manner. If there are no partial children in Pert\_i(X), then do nothing. Otherwise, for each partial child Y, merge the children of Y into the sibling lists of X so that the full or partial immediate sibling of Y in Pert\_i(X) is now an immediate sibling of the full endmost child of Y. Thus, the frontier of Pert\_i(X) still contains the same number of full leaves. Figure 4, below, illustrates the Reduce(Pert\_i(X)) operation.

![Diagram](image)

*Figure 4 - The Reduce(Pert\_i(X)) operation; (a) before Reduce(Pert\_i(X)); (b) after Reduce(Pert\_i(X))*

The basic templates introduced by Booth and Lueker were extended in [8] to include the case when the template matching process fails; a complete description was given using four additional templates describing when the template matching process fails. We follow the description of [4]. Let X be a Q-node of T. Now, if in every PQ-tree T' equivalent to the maximal PQ-
subtree rooted at \( X \), the number of maximal sequences of consecutive pertinent leaves in \( \text{Frontier}(T') \) is at least 2, then Template \( N_1 \) is matched. One of the situations that satisfies Template \( N_1 \) is shown below in Figure 5.

Let \( X \) be a Q-node of \( T \), where \( X \) is not the pertinent root. We say that \( X \) matches Template \( N_2 \) if \( X \) does not match Template \( N_1 \), and every PQ-tree \( T' \), equivalent to the maximal subtree rooted at \( X \), in which the pertinent leaves of \( \text{Frontier}(T') \) appear consecutively, both the leftmost and rightmost elements of \( \text{Frontier}(T') \) are empty. One of the situations that satisfies Template \( N_2 \) is shown below in Figure 6.

The templates which cater for non-planar cases when the node is a P-node are much easier. For Template \( N_3 \) to match we must have a P-node which is the pertinent root, with three or more partial children. Template \( N_3 \) is shown in Figure 7. For clarity we omit the rest of the partial children (if any), and the full and empty children (zero or more may be present) from the figure.
Template $N_4$ is similar to Template $N_3$, except that, in this case, the P-node in question may not be the pertinent root, and it must have at least two partial children. Figure 8, below, illustrates this template, where, again, we have omitted the rest of the partial children, and the full and empty children from the figure.

![Figure 8 - Template $N_3$](image)

We show in [4] that these templates are sufficient to describe all the cases for when the normal PQ-tree reduction process fails. Using the Templates $N_1$, $N_2$, $N_3$ and $N_4$ we may obtain the following algorithm which deals with a non-planar situation. For convenience we repeat, in comments in the algorithm, the requirements for every template to match.

**Algorithm 1: Place_Handles_Non_Planar_Graph(X)**

( The template matching in PQ-tree T has failed
  at a node X, we place handles to
  allow the reduction to proceed )

If Template $N_1(X)$
then    ( Q-node, two or more isolated sequences of full children )
  Place_Handle_N1(X)
else If Template $N_2(X)$
then    ( Q-node, not pertinent root, no full child endmost )
  Place_Handle_N2(X)
else If Template $N_3(X)$
then    ( P-node, pertinent root, three or more partial children )
  Place_Handle_N3(X)
else If Template $N_4(X)$
then    ( P-node, not pertinent root, two or more partial children )
  Place_Handle_N4(X)

283
end

We denote by $\gamma_g(G)$, the number of handles which we place to embed $G$. Before we begin the reduction for vertex $2$, we assume that we are to embed $G$ on the sphere, i.e. $\gamma_g(G) = 0$.

We consider each of the cases in turn. Note that the algorithms may place a number of handles. Suppose we are reducing for vertex $i$. At first we shall only place the pseudo-vertices. The base can only be determined at the end of the reduction for vertex $i$. Suppose we have determined that the full leaves from a sequence $\text{Pert}_j(X)$ of pertinent children of $X$ must be deleted. When we place a pseudo-vertex for some of the sequences $\text{Pert}_j(X)$, we shall mark the pertinent leaves which are in $\text{Pert}_j(X)$ as $\text{void}$. This marking process removes all references to the sequence of pertinent leaves from the Q-node $X$. This will then allow the reduction algorithm to continue the template matching process.

For the rest of this section, in the diagrams used to illustrate the different cases we encounter, we will use black circles to denote the ends of the handles which we insert, and hence the pseudo-vertex which we place.

Template $N_1$ is matched for a Q-node $X$, when we have at least two maximal subsequences of pertinent children in any $\text{Frontier}(T')$, where $T'$ is equivalent to the maximal subtree of $T$ rooted at $X$. The first operation we perform is $\text{Reduce}(\text{Pert}_j(X))$ for every $\text{Pert}_i(X)$. Thus, our maximal subsequences of pertinent children are now only full children. We have two subcases, depending on whether $X$ is the pertinent root or not.

**Case 1:** Suppose that $X$ is the pertinent root. Then, we select some subsequence $\text{Pert}_i(X)$. At the end of the reduction we will identify the edges represented by the pertinent leaves in the frontier of $\text{Pert}_i(X)$ in the base. For all other subsequences $\text{Pert}_j(X)$, we identify the pertinent leaves to a pseudo-vertex, and mark these leaves void.

**Case 2:** Suppose that $X$ is not the pertinent root. In this case, if a full child is endmost, we select a sequence $\text{Pert}_i(X)$ containing that child. For all other sequences $\text{Pert}_j(X)$, we identify the pertinent leaves to a pseudo-vertex, and mark these full leaves void. We do not identify the sequence $\text{Pert}_i(X)$ to a
pseudo-vertex, because the reduction is now able to proceed without the placement of a further handle. In this way we attempt to minimise the number of handles placed.

Algorithm 2, below, gives the full algorithm.

**Algorithm 2:** Place Handle N_1(X)

(Place Handle(s) to enable reduction to continue after reduction failed and X matched Template N_1)

(Q-node, two or more isolated full sequences of children)
(see Figure 9(a))

Let the maximal sequences of pertinent children be
Pert_1(X), Pert_2(X), ..., Pert_k(X); where k \geq 2

For all sequences Pert_i(X) do
  Reduce(Pert_i(X))

If X is not the pertinent root
  then
    If any full child Y of X is endmost
      then
        let Pert_i(X) be the sequence to which that
        full child belongs
      else
        Pert_i(X) = ∅
    else
      (X is the pertinent root)
      Select any sequence Pert_i(X)

      (Pert_i(X) now contains a sequence of full children from which we
do not want to place a handle)

for all sequences Pert_j(X) ≠ Pert_i(X) do
  Identify Pert_j(X) in a pseudo-vertex
  γ_g(G) = γ_g(G) + 1
  Mark Pert_j(X) void (i.e. delete Pert_j(X))

(see Figure 9(b))
(at this stage $X$ may be matched to one of Templates $P_0, \ldots, P_6, Q_0, \ldots, Q_3$)

If $X$ is not the pertinent root
then
\begin{itemize}
  \item Continue reduction and apply Template $Q_2$
  \item or Template $Q_0$ to $X$
\end{itemize}
depending if $\text{Pert}_i(X) \neq \emptyset$ or not
else
\begin{itemize}
  \item Continue reduction and apply Template $Q_2$
  \item or Template $Q_3$ to $X$,
\end{itemize}
depending if $\text{Pert}_i(X)$ is endmost or not
end

In Figure 9, we show a possible situation for some 2-connected non-planar graph $G$ when we are reducing for vertex 17.

![Diagram](a) ![Diagram](b)

*Figure 9 - Insertion of Handles when Template $\mathcal{K}_1$ is matched*

(a) Before reduction for vertex 17; (b) after placement of pseudo-vertex

286
We may now consider Algorithm Place_Handle_N2(X), which caters for the case when the reduction process fails, Template N1 does not match, and Template N2 is matched. Since we have all pertinent children appear in one maximal sequence Perti(X), it is sufficient to place a single handle. Again, we perform Reduce(Perti(X)). Then, we identify all the pertinent leaves to a single pseudo-vertex. Algorithm 3, below, gives the full algorithm.

**Algorithm 3: Place_Handle_N2(X)**

{ Place Handle(s) to enable reduction to continue after
  reduction failed, Template N1 does not match,
  and X matched Template N2}

{ Q-node, not pertinent root, no full child endmost }
{ see Figure 10 }

Let the maximal sequence of pertinent children be Perti(X)
Reduce(Perti(X))
Identify Perti(X) in a pseudo-vertex
\[ \gamma_g(G) = \gamma_g(G) + 1 \]
Mark Perti(X) as void

{ see Figure 10(b) }
{ at this stage X may be matched to Template Q0 }
{ since all pertinent children have been deleted }
Apply Template Q0 to X

end

Figure 10, below, gives the diagram illustrating Template N2. Once again, the psuedo-vertex is represented by a black circle, and we are reducing for vertex 17.
The rest of the algorithms deal with the Templates $N_3$ and $N_4$, which apply when $X$ is a P-node. Algorithm Place_Handle_3 uses a variation of the replacement pattern for Template $P_6$. Suppose a P-node $X$ satisfies Template $N_3$, so $X$ must be the pertinent root. Further, suppose that there are $k$ ($k > 2$) partial children of $X$. Let $P_l$ and $P_m$ be any two partial children of $X$. We place the full children of $X$ between $P_l$ and $P_m$. We then merge $P_l$ and $P_m$ to create a new Q-node $Y$ with both endmost children empty, and all full children appearing in one consecutive subsequence $Pert_i(Y)$ of the children of $Y$. We then pair off the other partial children, except for possibly one, and merge each pair, thereby reducing the number of subsequences of full children to $\left\lceil \frac{k}{2} \right\rceil$. For every partial child $W$ and corresponding maximal subsequence $Pert_j(W) \neq Pert_i(Y)$, we identify the pertinent leaves of Frontier($W$) in a pseudo-vertex, and mark all full children void. At the end of the reduction, $Pert_i(Y)$ is identified as our base. Algorithm 4, below, gives Algorithm Place_Handle_3.
Algorithm 4: Place_Handle_N3(X)

(Place Handle(s) to enable reduction to complete after reduction failed and X matched Template N3)

(P-node, pertinent root, three or more partial children)
(see Figure 11)

Let the partial children be P₁, P₂, ..., Pₖ; where k > 2
Denote the full and empty endmost children of any Pᵢ by
    Full(Pᵢ) and Empty(Pᵢ)

Let ℙ be a partition of the partial children into \[ \left\lfloor \frac{k}{2} \right\rfloor \] sets, so that
    each set, except for possibly one set (if k is odd), has two elements

Select one of these sets with two partial children Pᶠ and Pᵐ
Remove any full children of X and place them
    as children of a new P-node Y

Join Pᶠ and Pᵐ together as follows
    If Y has children
        then
            Add Y as an immediate sibling of
                Full(Pᶠ) and Full(Pᵐ)
            Let new endmost child of Pᶠ be Empty(Pᵐ)
        else
            Let Full(Pᶠ) and Full(Pᵐ) be immediate siblings
            Let new endmost child of Pᶠ be Empty(Pᵐ)
    Denote full sequence of children of Pᶠ by Main_Group

For every element of ℙ that contains a pair Pᵢ and Pⱼ (i, j ≠ ℓ, m) do
    Let Full(Pᵢ) and Full(Pⱼ) be immediate siblings
    Let new endmost child of Pᵢ be Empty(Pⱼ)
    Delete Pⱼ
    Identify full children of Pᵢ in a pseudo-vertex
    \[ γ_g(G) = γ_g(G) + 1 \]
Mark full children of \( P_i \) as void

if \( k \) is odd

then

Identify \( P_k \) in a pseudo-vertex

\[ \gamma_g(G) = \gamma_g(G) + 1 \]

Mark full children of \( P_k \) as void

end

Figure 11, below, shows a subgraph of a non-planar graph \( G \), when we are reducing for vertex 17.

\[ \text{Figure 11 - Illustration of Handles when Template N}_3 \text{ is matched} \]

(a) Before reduction for vertex 17; (b) after placement of pseudo-vertex

When Template \( N_4 \) is matched, the situation is almost the same as for Template \( N_3 \). Suppose a P-node \( X \) matches Template \( N_4 \), so \( X \) is not the
pertinent root. Further, suppose that there are \( k \) (\( k > 1 \)) partial children of \( X \). Let \( P_1 \) and \( P_m \) be any two partial children of \( X \). The full children of \( X \) must be placed between \( P_1 \) and \( P_m \). We then merge \( P_1 \) and \( P_m \) to create a new Q-node \( Y \) with both endmost children empty, and all full children appearing in one consecutive subsequence \( \text{Pert}_i(Y) \) of the children of \( Y \). We then pair off all, except for possibly one, of the remaining partial children and merge those partial children that were paired off, thereby reducing the number of subsequences of full children to \( \left\lceil \frac{k}{2} \right\rceil \). For every partial child \( W \) which does not have an endmost full child, and corresponding maximal subsequence \( \text{Pert}_i(W) \), we identify the pertinent edges of \( \text{Frontier}(W) \) in a pseudo-vertex, and mark all full children void. Note that we avoid placing an extra handle if \( k \) is odd. If this situation occurs, then there is some sequence \( \text{Pert}_i(X) \) which has a full endmost child. We do not identify the sequence \( \text{Pert}_i(X) \) to a pseudo-vertex, because the reduction is now able to proceed without the placement of a further handle. Again, we attempt to minimise the number of handles placed.

Algorithm 5, below, details the algorithm Place_Handle_N4.

**Algorithm 5**: Place_Handle_N4(X)

\( \text{(Place Handle(s) to enable reduction to continue after reduction failed and X matched Template N4)} \)

\( \text{(P-node, not pertinent root, two or more partial children)} \)

\( \text{(see Figure 12)} \)

Let the partial children be \( P_1, P_2, ..., P_k; \) where \( k \geq 2 \)

Denote the full and empty endmost children of any \( P_i \) by

Full(\( P_i \)) and Empty(\( P_i \))

Let \( \mathcal{P} \) be a partition of the partial children into \( \left\lceil \frac{k}{2} \right\rceil \) sets, so that

each set, except for possibly one set (if \( k \) is odd), has two elements

For every element of \( \mathcal{P} \) that contains a pair \( P_i \) and \( P_j \) (\( i, j \neq \ell, m \)) do

Let Full(\( P_i \)) and Full(\( P_j \)) be immediate siblings

291
Let new endmost child of $P_i$ be $\text{Empty}(P_j)$
Delete $P_j$
Identify full leaves from $P_i$ in a pseudo-vertex
\[ \gamma_{g}(G) = \gamma_{g}(G) + 1 \]
Mark full children of $P_i$ as void

If $k$ is odd
then
   Apply Template $P_5$ to $X$
else
   Apply Template $P_3$ to $X$
end

Figure 12, below, illustrates the situation when Template $N_4$ is matched.

---

Figure 12 - Illustration of Handles when Template $N_4$ is matched
(a) - Before reduction for vertex 17; (b) after placement of pseudo-vertex
All that remains to be discussed is the actual placing of the second end of the handles which we insert, and to prove the algorithm correctness. The placing of the handles inserted during the reduction for a vertex \( i \) is done once the reduction process for vertex \( i \) has been completed. We have the following lemma.

**Lemma 1:** Not every sequence of pertinent children during a reduction for a vertex \( i \) is marked void.

**Proof:** First of all, note that, for a node \( X \), Templates \( N_2 \) and \( N_4 \) do not allow \( X \) to be the pertinent root. If Template \( N_1 \) is matched, and \( X \) is the pertinent root, then Algorithm 2 explicitly selects a sequence \( \text{Pert}_i(X) \) not to be marked void. If Template \( N_3 \) is matched, then we join two partial children, and the pertinent leaves from those two partial children, as well as the full children of \( X \) are not marked void.

Using Lemma 1 we may identify the edges represented by full leaves not marked void in the base. Now, we place a handle from each pseudo-vertex created during the reduction for vertex \( i \) to the base, and place the corresponding identified edges along that handle.

Repeated application of the PQ-tree reduction algorithm as described in Algorithm A.2, together with the vertex addition for a graph has complexity \( O(q) \) (see [3] and [4] for more details). We call this algorithm the Pruned Reduction Algorithm. The complexity of Algorithm 1 can now easily be derived. The only detail that needs to be elaborated on is the complexity of \( \text{Reduce}(\text{Pert}_i(X)) \) operation for all maximal subsequences, \( \text{Pert}_i(X) \), of pertinent children of a Q-node \( X \). To successfully perform \( \text{Reduce}(\text{Pert}_i(X)) \), we traverse the children of \( X \) from left to right, looking for maximal sequences of pertinent children. It is shown in [4] that \( \text{Reduce}(\text{Pert}_i(X)) \) has complexity \( O(\text{Children of } X) \). We may scan these children \( O(p) \) times during a reduction pass. From [7] it is shown that \( O(\text{Children}(X)) = O(p) \) as well. So, during a reduction pass for a vertex, \( \text{Reduce}(\text{Pert}_i(X)) \) has complexity \( O(p^2) \). Therefore, overall, the complexity of \( \text{Reduce}(\text{Pert}_i(X)) \) is \( O(p^3) \), because there are \( p \) reduction passes. It is not hard to see that all other operations only consider pertinent nodes during the reduction, and
since there are $O(q)$ pertinent nodes in total (see [3] or [4] for more details), from [4] we have the following result on Algorithm 1.

**Theorem 1**: The Pruned Reduction Algorithm, together with Algorithm 1, embeds a 2-connected graph $G$ on a surface of genus $\gamma_g(G)$, and has $O(p^3 + q)$ time complexity.

**Conclusion**
Because there is no efficient algorithm for finding the genus of a graph, in general the results of the algorithm are not easy to compare against the actual genus of the graph. An embedding of a 2-connected graph $G$ on a surface of genus $n$ is a 2-cell embedding if a closed curve in a region of this embedding can be continuously deformed (on the surface) to a single point. The maximum genus $\gamma_m(G)$ of a 2-connected graph $G$ is the maximum genus of a surface on which $G$ can be 2-cell embedded. The maximum genus of a graph is defined (see [2]). There are two well-known types of graphs for which the genus $\gamma(G)$ and maximum genus $\gamma_m(G)$ is known. The graphs are the complete graph on $p$ vertices, $K_p$, and the complete bipartite graph with partite sets $m$ and $n$, namely $K_{m,n}$. The following results were obtained on the complete graph $K_p$, on $p$ vertices.

<table>
<thead>
<tr>
<th>$p$</th>
<th>$\gamma(K_p)$</th>
<th>$\gamma_g(K_p)$</th>
<th>$\gamma_m(K_p)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>9</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 1

For the bipartite graphs $K_{3,n}$ we obtained the following results.
<table>
<thead>
<tr>
<th>n</th>
<th>$\gamma(K_{3,n})$</th>
<th>$\gamma_g(K_{3,n})$</th>
<th>$\gamma_m(K_{3,n})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2

Lastly, we observe that the Pruned Reduction Algorithm together with Algorithm 1 does not necessarily produce a 2-cell embedding of $G$. Let $H$ be the plane subgraph of $G$ obtained by deleting all the edges of $G$ that were ever marked void. Then, it may happen that we place, for two different reductions $i$ and $j$, two handles from the same two regions with respect to the embedding of $H$. Consider Figure 13, below, with two edges on two handles, namely the edges $e_1 = \{11, 17\}$ and $e_2 = \{13, 18\}$.

![Figure 13 - An example of an embedding of $G$ that is not a 2-cell](image)

It is easy to see that the embedding of $G$ in Figure 13 is not a 2-cell, if we place a closed curve $C$ along both handles which then cannot be shrunk to a single point.

It remains an open problem to determine if Algorithm 1 may be extended to produce 2-cell embeddings of a non-planar graph $G$ on some surface. Once a 2-cell embedding of $G$ is guaranteed, then it is an easy extension of the
basic embedding algorithm by [5] (which also uses PQ-trees) to provide an embedding of $G$ on the surface of genus $\gamma_g(G)$. Hence it provides a description of the corresponding circuit on a surface of reasonably low genus.

References


Appendix

This introduction to PQ-trees is of necessity short. For further reference, please refer to [3] or [4].

Given any edge \( e = (s, t) \) of a 2-connected graph \( G \), a 1-1 function \( f : V \rightarrow \{1, 2, ..., p\} \) is called an \textit{st-numbering} of \( G \) if

(a) \( f(s) = 1 \)
(b) \( f(t) = p \)
(c) For every vertex \( v \in V(G) - \{s, t\} \) there exists vertices \( u \) and \( w \) adjacent with \( v \) such that \( f(u) < f(v) < f(w) \).

Note that every 2-connected graph has an st-numbering [9]. Let \( H \) be a subgraph of a graph \( G \) such that the highest st-number of a vertex of \( H \) is \( k \), and \( |V(H)| = k \). The st-numbering provides an ordering of the vertices of a \( G \), such that, if \( G \) is a plane graph, then all edges from vertices of \( V(H) \) to vertices of \( V(G) - V(H) \) lie in one face of \( H \) [9].

Let \( \mathcal{U} = \{a_1, a_2, ..., a_m\} \) be a universal set. Then the class of \textit{PQ-trees} over the set \( \mathcal{U} \) is defined to be the set of all rooted trees whose leaves are elements of \( \mathcal{U} \) and whose internal vertices are either 'P-nodes' or 'Q-nodes' where a P-node is a node whose children may be permuted freely amongst themselves and a Q-node is a node whose children remain in a fixed order, although the order in which they appear in any planar realisation of the PQ-tree may be reversed. The properties of the P-nodes and Q-nodes are shown in the way we draw them in the PQ-tree. For a P-node, we use a circle which represents freedom of order, whereas a Q-node is represented by a rectangle which portrays the fact that the order amongst the children is fixed. As for the drawing of normal trees, we draw the children of a node below the node in question (thus the root is at the top of the diagram). When it is not important whether a node is a P-node, Q-node or a leaf, then we shall represent it by a triangle. Figure A.1 shows an example of a PQ-tree.
Furthermore, a PQ-tree is called *proper* if we place the following restrictions on P-nodes and Q-nodes:

(i) Every element $a_i$ appears exactly once as a leaf.

(ii) Every P-node has at least two children.

(iii) Every Q-node has at least three children. (This convention is adopted because there is no distinction between P-nodes and Q-nodes with less than three children.)

From now on we shall consider only proper PQ-trees. Consider a PQ-tree $T$. We define the *frontier* of $T$ to be the leaves of $T$ as read from left to right in the embedding. An *equivalence transformation* on a PQ-tree node either:

(i) Arbitrarily permutes the children of a P-node; or

(ii) Reverses the children of a Q-node.

We say that two PQ-trees $T_1$ and $T_2$ are *equivalent* if and only if the $T_1$ can be transformed via a series of zero or more equivalence transformations to $T_2$. If two PQ-trees are equivalent, then we write $T_1 \equiv T_2$. We note that this is an equivalence relation. For example, for the PQ-tree $T$ in Figure A.1, there are 64 trees in the equivalence class. Every two PQ-trees in the same equivalence class have different frontiers.

It is important to note that a PQ-tree $T$ of a universal set $\mathcal{U}$ is a description of the "allowable" permutations of $\mathcal{U}$. The frontier of every PQ-tree $T' \equiv T$
represents a valid permutation. Given a universal set $\mathcal{U}$, there is a spectrum of possible PQ-trees. At the ends of the scale, we have the null tree and the universal tree. The null tree is the empty PQ-tree, which is a PQ-tree without any nodes or leaves. Formally, the null tree is not actually a PQ-tree, but we include it for the sake of completeness. The null tree represents the most restricted PQ-tree - one in which we do not know anything about the allowable structure of the universal set $\mathcal{U}$. The set of valid permutations of the null tree is empty. The universal tree is the most unrestricted PQ-tree - a single P-node with children $a_i$ for all $a_i \in \mathcal{U}$. Thus the universal tree represents every possible permutation on the universal set $\mathcal{U}$. These two trees are shown below in Figure A.2.

![Null tree and Universal tree](image)

*Figure A.2: The Null tree and Universal tree*

Turning to Q-nodes, there are always two endmost children. These are the children which are always at the ends of the sequence of children of the Q-node (irrespective of reversal). The rest of the children are interior. Furthermore, we say that an immediate sibling of a Q-node's child is a child of the Q-node which appears adjacent to that node in every PQ-tree $T'$ such that $T' \equiv T$. Conceptually, the immediate siblings of a node are the neighbours of that node in $T$, the embedding of $T$. Thus, every interior node has exactly two immediate siblings and every endmost child has exactly one immediate sibling. The two endmost children will be useful for processing the Q-nodes in the PQ-tree efficiently, as we shall see later. Figure A.3 illustrates these terms.
We now discuss the general application of PQ-trees. Let $\mathcal{S}$ be a class of subsets of a universal set $\mathcal{U}$ (note that the elements of $\mathcal{S}$ need not be disjoint). Consider the following problem $\mathcal{P}$: Determine all permutations $\pi$ on $\mathcal{U}$ so that for every $S \in \mathcal{S}$, the elements of $S$ appear consecutively in $\pi$. Algorithm A.1 gives a formal description of a method which solves $\mathcal{P}$.

**Algorithm A.1:** Reduction ($\mathcal{U}, \mathcal{S}$)

\[
\text{find all permutations } \pi \text{ of } \mathcal{U} \text{ such that, for every set } S \in \mathcal{S}, \\
\text{all elements of } S \text{ appear consecutively in } \pi \\
\]

\[
\Pi = \{ \pi \mid \pi \text{ is a possible permutation of } \mathcal{U} \} \\
\text{For every } S \in \mathcal{S} \text{ do} \\
\quad \Pi = \Pi \cap \{ \pi \mid \text{all objects of } S \text{ are consecutive within } \pi \} \\
\quad \text{[ the reduction phase ]} \\
\text{end}
\]

We now describe how the PQ-tree data structure can be used to implement this reduction efficiently. We begin with a universal tree $T$ whose leaves are the elements of $\mathcal{U}$. Note that at this stage the trees equivalent to $T$ represent all permutations of the elements of $\mathcal{U}$. The inner loop of Algorithm A.1 is actually a modification procedure which adjusts $T$ to reflect the new constraint, namely that for each $S \in \mathcal{S}$ the elements of $S$ appear consecutively in the frontier of $T$. Effectively we are reducing the number of trees equivalent to $T$ (i.e. the set $\Pi$). We say that $T$ is $\mathcal{S}$-reduced if, for every $T' \equiv T$, the elements of $S$ appear consecutively in Frontier($T'$). We only need a single operation on $T$, denoted by Reduce ($T, S$), which modifies $T$ so that it becomes $S$-reduced. Informally, the process of performing such a reduction consists of scanning the PQ-tree node by node, starting from the leaves, and then looking for patterns in the structure of the subtree at each node, and
structurally modifying the subtree rooted at the node with a replacement. We call each pair of pattern and replacement at a node a Template. We now provide a formal description of this reduction.

Algorithm A.2: Reduce (T, S)

{ Constrain the PQ-tree T so that S appears consecutively in Frontier(T) }

QUEUE = empty
{ Queue contains nodes which can be matched and then replaced }

for each leaf \( \in U \) do
Add_to_Queue (leaf)

while not Finished do { Finished when all elements of S consecutive }

Current = Head of Queue
Match Templates to Current
if a Template matches
then Replace subtree rooted at Current
with replacement pattern
else
T = Null Tree
Halt

if \( S \subseteq \{ \text{Leaf} \} \) Leaf is a leaf of the subtree of T rooted at Current
then Finished
{ we have arranged all leaves of S }
{ consecutively in Frontier(T) }

else
if parent of Current_Node has all its children queued
then Add parent of Current to Queue

end

Notice that the changes are local, we only modify the node and its children. Also, the pattern which is matched depends only on the node and its children. We match the children of a node before we match the node itself.

There are three possible states a child of a node can be in. Consider any child \( X \). If none of the descendants of \( X \) which are leaves are in \( S \), we say that \( X \) is empty. If all of the descendants of \( X \) which are leaves are in \( S \), we say \( X \) is full, and lastly if some of the descendants of \( X \) that are leaves are in \( S \), but some are not, then we say \( X \) is partial. The templates matchings look for a combination of the node type, i.e. whether the node is a P-node or a Q-node or a leaf, and the states of its children. A node is said to be pertinent if some or all of its children are either full or partial with respect to \( S \). The pertinent subtree of \( T \) with respect to \( S \), denoted Pertinent(T,S), is the unique
subtree, rooted at a vertex X, of T of minimum height whose frontier entirely contains S. The root of the pertinent subtree is denoted by Root(T,S). For example, in Figure A.4, below, we show the pertinent subtree of the PQ-tree given in Figure A.1 when S = {a, c, e, f}. Intuitively we should only have to perform equivalence transformations on Pertinent(T,S).

![Diagram](image)

*Figure A.4 Pertinent(T,S) when S = {a, c, e, f}*

After matching each template and performing the appropriate replacement, we need to ensure that there is no information loss. In other words, the PQ-tree Reduce(T,S) must represent exactly every possible valid permutation of the frontier of T which has all elements of S appearing consecutively. We shall now describe all possible templates and replacements for a PQ-tree T. The following convention is adopted. Full nodes are shaded and partial nodes have only the right-hand side of the node shaded. Empty nodes are left unshaded. Similarly, whenever a child's node type does not matter, we shall represent it by a shaded or unshaded triangle. Furthermore, we always arrange the children of a partial node X in T so the pertinent descendants which are leaves of the maximal subtree rooted at X appear consecutively in Frontier(T). There are a number of templates, and we shall assign them two letter names. We will see next that there are seven templates for P-nodes (named P₀, P₁, ..., P₆), four templates for Q-nodes (named Q₀, Q₁, ..., Q₃) and two leaf templates L₀ and L₁.

Consider the leaves of T. Clearly there are only two templates, either a leaf is not in S, or it is in S. The replacement pattern is simply the same node unchanged except to note that the node is empty or full.
We consider P-nodes and Q-nodes separately. For P-nodes there are seven templates. Firstly, there are the cases where the children are either all empty or all full. Figures A.5 a and b, below show these cases respectively, giving Templates \( P_0 \) and \( P_1 \). Note that for Template \( P_1 \) the replacement is exactly the same as the original, except that the state of the node is now determined. This way we do not restrict the freedom of elements within \( S \) to permute amongst themselves.

![Pattern and Replacement](image)

\( (a) \) Template \( P_0 \)

\( (b) \) Template \( P_1 \)

*Figure A.5: Templates \( P_0 \) and \( P_1 \)*

The next case to consider is the one where some of the children are full and some are empty. This case we split into two further cases. When the current node we are at is \( \text{Root}(T,S) \), then we must cluster all the full children together, but still allow the full children, as a cluster, and the empty children to permute freely. Figure A.6 shows Template \( P_2 \). Notice that the replacement does not restrict the permutations any more than necessary; we allow the cluster of full nodes to permute, as a cluster, amongst the empty children and for full children to permute freely amongst themselves.
The second sub-case to consider occurs when we have some full children and some empty children at a node which is not Root(T,S). We partition the children into two groups, namely the full children and the empty children, and allow them to permute freely within the same cluster. This is in contrast to Template P₂ where the cluster of full children could permute with the empty children. The same situation is not allowed to exist here because we know that there are other elements of S elsewhere in the pertinent tree (or else we would be at Root(T, S) and Template P₂ would be used). A permutation with empty children on either side of the full group of children would not allow the other full children elsewhere in the pertinent tree to be adjacent to these full children. We label the node partial. Figure A.7 shows this template, Template P₃.
Notice that we change the node type to a Q-node, this is done to simplify the template definitions later. Although the PQ-tree is no longer proper (since the replacement Q-node only has two children), this situation will be rectified later in the template matching process. In the special cases where there is only one full child or one empty child, Figure A.8 shows the replacements for the Template $P_3$.

\[\text{Figure A.8: Special Replacement cases for } P_3\]

Now consider the case where at least one of the children is partial. As can be seen from the rest of the templates, only a Q-node may be partial. The first case is a special case where there is exactly one partial child, possibly some full children and possibly also some empty children. Again, this breaks into two special cases, where the node in question is $\text{Root}(T,S)$ and where it is not $\text{Root}(T,S)$. Consider the first case shown in Figure A.9.

Again, as in Template $P_2$ the full cluster may permute freely amongst the empty children. However, this time we must restrict the full (empty) children which were restricted to a fixed order in the partial Q-node to remain in that order. Note that the full children of the node may permute amongst themselves, but they must always be on one side of the Q-node (because we group all full children together).
The second sub-case is where the node in question is not Root(T,S) and we have exactly one partial child, no or possibly some full children and no or possibly some empty children. The same transformation is performed, except, as for Template P₃, we restrict the empty children of the node to permute freely only on one side of the Q-node. All fixed position empty and full children remain (necessarily) in their fixed positions. Template P₅ is shown below in Figure A.10.
The last case to consider for the P-nodes is when we have exactly two partial children. In this case, the node must be Root(T,S). If it is not, then there is no way we can continue the matching. To see this, consider the Template $P_5$ shown in Figure A.11, below. The full children of the node must be placed between the two partial nodes. The full children of the two partial nodes must be adjacent to the other full children. This means that the replacement has empty children on either side of the new Q-node. Hence if this node is not Root(T,S), then we cannot place other full elements from the rest of the pertinent tree adjacent to these full elements.
We shall now consider the templates for Q-nodes. The Templates $Q_0$ and $Q_1$ are the same as for P-nodes. They cater for Q-nodes whose children are all empty or all full, respectively. Figure A.12 shows Templates $Q_1$ and $Q_0$.

Compared to the corresponding P-node cases, matters are much simpler for the rest of the Q-node templates. We can condense the Q-node versions of Templates $P_2$, $P_3$, $P_4$ and $P_5$ into one Template $Q_2$. This template is valid for the pattern shown in the Figure A.13.
The left to right order of the node's children must be exactly as shown, but some of the information may be missing (we delete the corresponding part from the replacement). There can be up to one partial child, and if present, it must be as shown in Figure A.13; i.e. it must be adjacent to the sequence of full children (if any full children are present), or if no full children are present then it must be an endmost child. Either the empty or the full children may be missing from the diagram. If the full children are present, then they must occupy one side of the Q-node in one cluster (i.e. a full child must be an endmost child). The last Q-node template is Template Q₃, and is for the case when the node is Root(T, S) and both endmost children of the node are not full. See Figure A.14 below.
Figure A.14: Template $Q_3$ for $\text{Root}(T,S)$ and at most two partial children

Note that this template can only apply to $\text{Root}(T,S)$, so exactly the same argument applies as for Template $P_6$. As in Template $Q_2$, we may have no partial children but here we can also have up to two partial children. If both partial children are present, then any full children must lie between the two partial nodes. Again, as in Template $Q_2$, either the full children or the empty children or both may be missing from the template.

Let us consider an example of the template matching process as applied to a graph planarity testing algorithm. Unfortunately, it is impossible to construct an example that uses all the templates. For example, Templates $P_2$, $P_4$, $P_6$ and $Q_3$ are mutually exclusive. Consider as an example the (non-planar) st-numbered graph $K_5$ given in Figure A.15.
Let $H$ be a subgraph of a graph $G$ such that the highest st-number of a vertex of $H$ is $k$, and $\lvert V(H) \rvert = k$. Recall that the st-numbering provides an ordering of the vertices of a $G$, such that, if $G$ is planar, then in an embedding of $G$ in the plane all edges from vertices of $V(H)$ to vertices of $V(G) - V(H)$ lie in one face of $H$. We start with the universal set $\mathcal{U}$ representing all edges directed out of vertex $1$. When we add the vertex with st-number $k$ ($2 \leq k < p$) to $H$, we remove all leaves of the PQ-tree representing edges directed into vertex $k$, and add to the PQ-tree a P-node with children being leaves representing edges directed out of vertex $k$. Suppose we are reducing for a vertex $k$. Since the PQ-tree represents all valid permutations of the edges from $H$ to $V(G) - V(H)$, if we are unable to find an equivalent PQ-tree $T$ which has all edges directed into vertex $k$ arranged consecutively in $\text{Frontier}(T)$, then this implies that the edges directed into vertex $k$ cannot be arranged consecutively in any embedding of $V(H) \cup \{k\}$. Thus we may conclude that $G$ is non-planar.

The initial PQ-tree $T$ is shown in Figure A.16, below. Note how $T$ represents all the edges, yet does not restrict them into any particular combination in a planar realisation of $G$. 

312
Now, we gather all leaves representing edges directed into vertex $v_2$. In this case there is only one such leaf. Thus we replace the leaf $(v_1,v_2)$ with a P-node having as children leaves representing the edges directed out of vertex $v_2$, namely the leaves $(v_2,v_3), (v_2,v_4)$ and $(v_2,v_5)$. See Figure A.17.

Now we consider the set $S = \{(v_1,v_3), (v_2,v_3)\}$. This time, the reduction is non-trivial. Template $L_1$ is triggered twice, once for each leaf element in $S$. The parent of leaf $(v_1,v_3)$ may not be matched yet, since not all of its pertinent children have been matched. However, we match the parent of $(v_2,v_3)$ to Template $P_3$. Consequently, we may now match Root$(T, S)$ (which, in this
case is Root(T) with Template $P_4$. Now we remove the sequence of full children from Reduce(T, S) and add in their place a P-node. To this P-node we add the children leaves $(v_3,v_4)$ and $(v_3,v_5)$. The resulting PQ-tree is shown in Figure A.18.

![PQ-tree and Graph](image)

*Figure A.18: After reduction is complete for vertex $v_3$*

Take the next set, namely $S = \{(v_1,v_4), (v_2,v_4), (v_3,v_4)\}$. Template $L_1$ is triggered three times, once for each leaf element in $S$. The parent of leaf $(v_1,v_4)$ may not be matched yet, since not all of its pertinent children have been matched. However, we match the parent of $(v_2,v_4)$ to Template $P_3$, and the parent of $(v_3,v_4)$ to Template $P_3$. At this stage no template matches the next node on the queue. Although Template $Q_3$ would match, the current node is not Root(T, S) and thus Template $Q_3$ may not be applied. Consequently the reduction process fails and we conclude that $K_5$ is non-planar. To see that the reduction process must fail at this stage is easy. Consider Figure A.19, below, which shows the PQ-tree $T$ and what the corresponding sub-graph would have looked like had we applied Template $Q_3$ to $T$. We observe that, in order to group $(v_1,v_4)$ with the other pertinent edges, we would have to cross other edges.
Figure A.19: PQ-tree after applying Template Q₃ incorrectly