Special Issue: SAICSIT '99
Contents

Preface
P. Machanick ............................................................... 1

Research Articles
Active Learning: Issues and Challenges for Information Systems and Technology
RD Quilling, GJ Erwin and O Petkova ............................................................... 5

A Generic Modelling Framework for Interactive Authoring Support Environments
Paula Kotze ............................................................... 15

O Petkova and JD Roode ............................................................... 26

An Information-Theoretic Semantics for Belief Change
T Meyer ............................................................... 33

A Complexity Metrics Model for Software Correction
A Törn, T Andersson and K Enholt ............................................................... 40

A Conceptual Design for High-Volume Data Processing of Warehouse Database into Multidimensional Database
Paisarn Trakulsuk and Vichit Avatchanakorn ............................................................... 49

A Pragmatic Approach to Bitemporal Databases: Conceptualization, Representation and Visualisation
Chiyaba Njovu and WA Gray ............................................................... 58

A Building Recognition System
SP Levitt and B Dwolatzky ............................................................... 68

Computer Programming and Learning to Write
John Barrow ............................................................... 77

Co-operating to Learn using JAD Technologies
TA Thomas ............................................................... 87

Critical Success Factors for the Implementation of DSS at a Selection of Organisations in Kwazulu/Natal
URF Averweg and GJ Erwin ............................................................... 95

Enhancing the Predictability of Two Popular Software Reliability Growth Models
Peter A Keiller and Thomas A Muzzuchi ............................................................... 105
Generalised Unification of Finite Temporal Logic Formulas
Scott Hazelhurst ................................................................. 110

Harmonizing Global Internet Tax: A Collaborative Extranet Model
E Lawrence and B Garner ...................................................... 119

Improving Object Oriented Analysis by Explicit Change Analysis
Lui Yu, Siew Chee Kong, Yi Xun and Miao Yuan ......................... 128

Reconciling the Needs of New Information Systems Graduates and Their Employers in Small, Developed Countries
Rodney Turner and Glenn Lowry ........................................... 136

Shortest Delay Scheduling Algorithm for Lossless Quality Transmission of Stored VBR Video under Limited Bandwidth
Fei Li, Yan Liu, Jack Yi-Bun Lee and Ishfaq Ahmad .................... 146

Software Croma Keying in an Immersive Virtual Environment
Frans van der Berg and Vali Lalioti ........................................ 155

Some Automata-Theoretic Properties of $\cap$-NFA
Lynette van Zijl and Andries PJ van der Walt ......................... 163

Statistical Analysis of an Automated Computer Architecture Learning Environment
JT Waldron, J Horgan and G Keogh ....................................... 168

The CILT Multi-Agent Learning System
Herna L Viktor ..................................................................... 176

The Development of a Generic Framework for the Implementation of Cheap, Component-Based Virtual Video-Conferencing System
Soteri Panagou and Shaun Bangay .......................................... 185

The Role of Experience in User Perceptions of Information Technology: An Empirical Examination
Meliha Handzic and Graham Low ............................................ 194

What are Web Sites Used for: Cost Savings, Revenue Generating or Value Creating?
Man-Ying Lee ...................................................................... 201

New Ideas Papers
Approaches to Video Transmission over GSM Networks
Bing Du and Anthony Maeder ................................................ 210

From Information Security Baselines to Information Security Profiles
Rossouw von Solms and Helen van der Haar ............................. 215

Grounded Theory Methodology in IS Research: Glaser versus Strauss
J Smit .................................................................................. 219

Introducing a Continuum of Abstraction-Led Hierarchical Search Techniques
Robert Zimmer and Robert Holte ......................................... 223
Multimedia as a Positive Force to Leverage Web Marketing, with Particular Reference to the Commercial Sector

Stan Shear ......................................................... 229

Understanding HCI Methodologies

Peter Warren ....................................................... 234

Electronically Published Papers

Experience Papers

A Java Client/Server System for Accessing Arbitrary CANopen Fieldbus Devices via the Internet

Dieter Bühler, Gerd Nusser, Gerhard Gruhler and Wolfgang Küchlin ......................................................... 239

An Object-Oriented Framework for Rapid Client-side Integration of Information Management Systems

Ralf-Dieter Schimkat, Wolfgang Küchlin and Rainer Krautter ................................................................. 244

Distributed Operating Systems: A Study in Applicability

Jürgen Prange and Judith Bishop ......................................................... 249

Formal Verification with Natural Language Specifications: Guidelines, Experiments and Lessons so far

Alexander Holt ......................................................... 253

Introducing Research Methods to Computer Science Honours Students

Vashti Galpin, Scott Hazelhurst, Conrad Mueller and Ian Sanders ................................................................. 258

Visualising Eventuality Structure

ST Rock ......................................................... 264

Electronically Published Papers

SAICSIT’99

South African Institute of Computer Scientists and Information Technologists

Annual Research Conference 17-19 November 1999

Prepare for the New Millennium

*Is there life after y2k?*

Mount Amanzi Lodge, Hartebeespoort, South Africa

Sponsors

[Logos of sponsors]

http://www.cs.wits.ac.za/

PHDS research/PHDS.html

Think different.
Preface

Philip Machanick, Overall Chair: SAICSIT'99

Running SAICSIT'99, the annual research conference of the South African Institute for Computer Scientists and Information Technologists, has been quite an experience.

SAICSIT represents Computer Science and Information Systems academics and professionals, mainly those with an interest in research. When I took over as SAICSIT president at the end of 1998, the conference had not previously been run as an international event. I decided that South African academics had enough international contacts to put together an international programme committee, and a South African conference would be of interest to the rest of the world.

I felt that we could make this transition at relatively low cost, given that we could advertise via mailing lists, and encourage electronic submission of papers (to reduce costs of redistributing papers for review).

The first prediction turned out to be correct, and we were able to put together a strong programme committee.

As a result, we had an unprecedented flood of papers: 100 submitted from 21 countries. As papers started to come in, it became apparent that we needed more reviewers. It was then that the value of the combination of old-fashioned networking (people who know people) and new-fashioned networking (the Internet) became apparent. While the Internet made it possible to convert SAICSIT into an international event at relatively low cost, the unexpected number of papers made it essential to find many additional reviewers on short notice. Without the speed of e-mail to track people down and to distribute papers for review, the review process would have taken weeks longer, and it would have been much more difficult to track down as many new reviewers in so little time.

Even so, the number of referees who were willing to help on short notice was a pleasant surprise.

The accepted papers cover an interesting range of subjects, from management-interest Information Systems, to theoretical Computer Science, with subjects including database, Java, temporal logic and implications of e-commerce for tax.

In addition, we were very fortunate in being able to invite the president of the ACM, Barbara Simons as a keynote speaker. Consequently, the programme for SAICSIT'99 should be very interesting to a wide range of participants.

We were only able to find place in the proceedings for 36 papers out of the 100 submitted, of which only 24 are full research papers. While this number of papers is in line with our expectation of how many papers would be accepted in each category, we did not have a hard cut-off on the number of papers, but accepted all papers which were good enough, based on the reviews. Final selection was made by myself as Programme Chair, and Derrick Kourie, as editor of the South African Computer Journal. Additional papers are published via the conference web site.

We believe that we have put together a quality programme, and hope you will agree.

Acknowledgments

I would like to thank the South African Computer Journal production team, Andries Engelbrecht and Herna Viktor, respectively from the Department of Computer Science and Informatics, University of Pretoria, for their work on producing the proceedings.

The reviewers listed overleaf did an excellent job: many wrote very detailed reports, sometimes after being called in on very short notice. Inevitably, there were some glitches resulting from the unexpected workload, but the buck stops with the programme chair: I promise to do better next time.

I would also like to thank my own department for putting up with the extra work and expense that running a conference entails. I tried not to burden them with too much extra work, but our secretaries, Zalm Gowar and Leanne Reddy, inevitably had to take on some extra work. John Ostrowick provided valuable assistance with design of our web pages and call for papers poster. Carol Kernick, who handles our finances and membership records, did a fine job of keeping up with the demands of the conference.

Finally, I would like to thank our sponsors, whose contribution made this conference been possible:

- PricewaterhouseCoopers - sponsored generous prizes and the conference banquet
- National Research Foundation (NRF) - provided financial support
- University of the Witwatersrand - provided financial support
- Programme for Highly Dependable Systems, University of the Witwatersrand - provided financial support
- Standard Bank - provided financial support
Editorial

- Apple Computer – provided equipment for the conference
- Qualica – provided technical support including helping with the conference web site

Web Site

For more information about SAICSIT, including a pointer to the conference site, see <http://www.saicsit.org.za>.

Referees

- Department of Computer Science, University of Pretoria
  - Derrick Kourie
  - Bruce Watson
  - Vali Lalioti
  - Andries Engelbrecht
  - Ivan Mphahlele
- German National Research Center for Information Technology - GMD
  - Gernot Goebbels
- School of Math Stat and IT, Natal University Pietermaritzburg
  - Peter Warren
- Graduate School of Business, University of Cape Town
  - Kurt April
- CSIR
  - James Jardine
- Department of Computer Science, Rhodes University
  - Shaun Bangay
  - Peter Clayton
  - John Ebden
  - Richard Foss
  - George Wells
  - Peter Wentworth
- Information Technology Division, Rhodes University
  - Caro Watkins
- Department of Information Systems, Rhodes University
  - Brenda Mallinson
  - Dave Sewry
- Department of Informatics, University of Pretoria
  - Carina de Villiers
  - Herna Viktor
  - Niek du Plooy
  - Elsie van Rooyen
  - Machdel Matthee
  - Alan Abrahams
  - Jackie Phahlamohlaka
- School of Information Technology & Engineering, University of Ottawa
  - Stan Szpakowicz
  - Dwight Makaroff
- Department of Information Systems, Victoria University
  - Glenn Lowry
  - Peter Shackleton
  - Tas Adam
  - Alastair Wallace
  - Stephen Burgess
  - Dave Burgess
  - Julie Fisher
  - Jerzy Lepa
  - Geoff Sandy
  - Rod Turner
  - Alastair Wallace
  - Andrew Wenn
- Peninsula School of Computing and Information Technology, Monash University
  - Ainslie Ellis
- School of Information Management and Systems, Monash University
  - Angela Carbone
- School of Computer Science & Software Engineering, Monash University
  - Dianne Hagan
  - Judy Sheard
- School of Management Information Systems, Deakin University
  - John Lamp
  - David Mackay
  - Philip Joyce
• Computer and Automation Research Institute, Hungarian Academy of Sciences
  - Ferenc Vajda
• Department of Computer Science, University of the Witwatersrand
  - Ian Sanders
  - David Lubinsky
  - Conrad Mueller
  - Yinong Chen
  - Bob Baber
  - Vashti Galpin
  - András Salamon
  - Scott Hazelhurst
  - Philip Machanick
  - Zoltán Fazekas
• Department of Electrical Engineering, University of the Witwatersrand
  - Barry Dwolatzky
  - Farzin Aghdasi
• Department of Mathematics, University of the Witwatersrand
  - Jonathan Burgess
• School of Law, University of the Witwatersrand
  - Victoria Bronstein
• Edward Nathan & Friedland Attorneys, Johannesburg
  - Justine White
• Department of Information Systems & Technology, University of Durban-Westville
  - Geoff Erwin
• Department of Computer Science, Clemson University, Clemson, SC
  - Karen Hay
• Logical SA
  - Philip Green
• Department of Computer Science, University of Bristol
  - Alan Chalmers
• Department of Systems and Computer Science, Howard University, Washington, D.C
  - Todd Shurn
  - John Trimble
• Department of Computer Science, University of Cape Town
  - Sonia Berman
• King's College, London
  - Peter Wood
• Department of Information Systems, University of Cape Town
  - Paul Licker
  - Steve Erlank
  - Alemayehu Molla
• Department of Department of Mathematics and Applied Mathematics, University of Cape Town
  - Ingrid Rewitzky
  - Peter Jipsen
  - Renkuan Guo
• Department of Information Systems, University of the Western Cape
  - Andy Bytheway
• Department of Electrical Engineering and Computer Science, The George Washington University, Washington, DC
  - Rachelle Heller
• School of Biophysical Sciences and Electrical Engineering, Swinburne University of Technology
  - Ian Macdonald
• Electrical and Electronic Systems Engineering, Queensland University of Technology
  - Anthony Maeder
• Computer Science Department, Åbo Akademi
  - Aimo Törn
  - Kaisa Sere
  - Mats Aspnas
• Division of Informatics, University of Edinburgh
  - Graeme Ritchie
  - Arturo Espinosa-Romero
  - Jon Oberlander
Editorial

- Chris Brew
- Alexander Holt

• Department of Computer Science and Engineering, Helsinki University of Technology
  - Tommi Junnila
  - Nisse Husberg

• Technical University of Budapest, Department of Measurement and Information Systems
  - Andras Pataricza

• NASA-Goddard Space Flight Center
  - Nigel Ziyad

• Department of Computer Science & Information Systems, University of Port Elizabeth
  - Janet Wesson
  - Leon Nicholls

• Computer Technology Department, Indiana University Purdue University Indianapolis
  - Tim Price

• Department of Electrical Engineering and Computer Science, The University of Michigan
  - Trevor Mudge

• Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
  - David Forsyth

• School of Computer Science, University of Birmingham
  - Mark Ryan

• Faculty of Mathematics, Computer Science, Physics & Astronomy, University of Amsterdam
  - Carlos Areces

• Department of Computer Science, Universidade do Vale do Rio dos Sinos - UNISINOS Rio Grande do Sul
  - Marcelo Walter

• School of Information Technology & Mathematical Sciences, University of Ballarat
  - Binh Pham
Generalised Unification of Finite Temporal Logic Formulas

Scott Hazelhurst

Department of Computer Science, University of the Witwatersrand, Johannesburg,
Private Bag 3, 2050 Wits, South Africa, scott@cs.wits.ac.za

Abstract

Given two temporal logic formulas, $g$ and $h$, we wish to know whether there is a modification, $m$, we can make to $h$ so that $g \Rightarrow m(h)$. This problem has important applications in hardware verification, where finding such modifications efficiently is important for an inferencing system. The temporal logic used, $TL$, is a relatively simple logic which is interpreted over finite state machine models. The modifications allowed are 'time-shifting' (nesting under the next-time temporal operator) and substitution of expressions for variables. An algorithm has been developed and implemented in the Voss verification system for the finite fragment of $TL$. The basis of the algorithm is to use a relatively compact, normalised representation of the formulas, which converts the problem into a generalised string matching problem. Some experimental results have been obtained from the use of the algorithm in practice.

Keywords: unification, temporal logic, pattern matching

Computing Review Categories: F2.2, F4.1, F4.2, I2.4

1 Introduction

Informally the problem to be solved is:

- Given two temporal logic formulas, $g$ and $h$, does $g \Rightarrow h$?

More generally,

- Is there a modification $m$ that can be made to $h$ so that $g \Rightarrow m(h)$. If so, what is the modification?

This paper addresses these problems. The rest of this section explains why they are interesting problems. Section 2 defines the logical framework (the syntax and semantics of the temporal logic, and the model in which it is interpreted), after which the problem examined is formally defined (e.g. what 'modified' means). Section 3 describes the initial algorithm used for a restricted form of the problem. Section 4 presents some real examples and experimental results.

Motivation

This problem arises in the implementation of an automatic verification system (called the VossProver). To see whether a program or circuit meets its specification, checking whether one formula implies another is a common task. The more general problem also occurs in this context, which is explained informally below.

The VossProver has a type of modus ponens rule: given a rule $h \Rightarrow p$, if we know that $h$ holds, then we can conclude that $p$ holds. Or, more generally if $g \Rightarrow h$, using the rule $h \Rightarrow p$, we can conclude that $p$ holds. ($\Rightarrow$ and $\Rightarrow$ capture different types of implication; this paper focuses on the former, the formal definition of which is given later).

Two ways in which temporal logic formulas can be modified is by time-shifting and substitution (these terms are explained in Section 2). Another VossProver inference rule is that if $h \Rightarrow p$ is a rule, then so is $m(h) \Rightarrow m(p)$, where $m$ is a modification made up of a time-shift and a substitution. So, if $g$ can be proved, and if $h \Rightarrow p$ is a rule, then if we can find a modification $m$ so that $g \Rightarrow m(h)$, then we can deduce that $m(p)$ holds (and this can, in turn, be used for further reasoning). For more details see [3, 4].

The problem posed in this paper is, in general, undecidable since we need to reason about arithmetic. However, it is possible to build decision routines that are robust. The criteria used to judge a decision procedure are:

- It must be sound: if the decision routine says $g \Rightarrow h$, then this must be true. False positives can't be tolerated.
- It very desirable that the routine can handle a large number of formulas that arise in practice. False negatives are undesirable but can be tolerated provided they happen rarely.
- The routine must be efficient. The decision routine may be called hundreds or even thousands of times in one verification run, and so good performance is critical for the verification tool to be used.

There are some trade-offs between these goals, and it will be possible to judge these criteria better when we have seen the theory and the examples.

2 The Logical Framework

This section presents the logical framework used in this paper. First, the model in which the logic is interpreted is briefly described. This is followed by a description of the
syntax and semantics of the temporal logic, after which explanations of time-shifts and substitutions are given. The final part of this section defines the goal of the paper formally and precisely.

2.1 The model

As background, the application area of this problem is hardware verification. Circuits are described at the gate or switch level, and finite state machine models are automatically extracted from the hardware descriptions. These finite state machine (FSM) models are accurate models of the circuit behaviour, modelling functional, timing and even some electrical behaviour. The state spaces are extremely large: examples with well over $2^{100000}$ states have been considered. More details can be found in [4].

For the purpose of this paper, we can consider the FSM to have two components: the set of states $S$ which is used to model the state of the circuit, and a (deterministic) next state function $Y : S \rightarrow S$, which models its behaviour. (In addition, there is an information ordering defined on $S$ which is used in the verification algorithm. However, this information ordering is not relevant to this paper; the purpose of mentioning this is to show that the technical setting is fairly general.)

2.2 The temporal logic TL

An FSM is used to model a circuit’s state and behaviour. The specification for the circuit is given as a set of assertions in the temporal logic, TL. This is a simple temporal logic, but one that has been used in many case studies. A general introduction to temporal logic can be found in [10] and a more rigorous introduction to TL can be found in [4].

TL is constructed from a set of predicates about the state space using logical operators (like conjunction), and temporal operators. For ease of presentation, first the scalar version of the logic (no variables) is presented, and then the extension to the full logic is given.

2.2.1 Definition of the logic

Let $G$ be a set of predicates over the state space: if $p \in G$ it is of type $p : S \rightarrow Q$, where $Q$ is the truth domain. Several possible truth domains are possible. For the purpose of this paper, think of $Q$ as being the booleans $\{f, t\}$. In the setting of [4], a four-valued truth domain is used. What follows holds for both truth domains. The syntax of the temporal logic is given below.

Definition 1. The set of scalar TL formulas is defined by:

$\text{TL} ::= G \mid \text{TL} \land \text{TL} \mid \neg \text{TL} \mid \text{Next TL} \mid \text{TL Until TL}$

Predicates in $G$ are used to describe or check the state of the system at a particular instant in time. To make this concrete, in our circuit domain, typical predicates might be

- $\lbrack \text{Clk} \rbrack = t$: Is the clock high?
- $\lbrack \text{Reset} \rbrack = L$: Is the reset line low?
- $\lbrack \text{Score} \rbrack < 16$: Is the value of group of lines identified by Score, when considered as a bit-vector, less than 16?

These predicates can be connected using conjunction and negation. Disjunction and implication are defined using De Morgan’s laws.

A predicate asks about the state of the machine at a particular instant in time. The temporal operators allow us to reason about the behaviour of the FSM over time. A temporal logic formula therefore is interpreted with respect to a sequence of states. First, if we consider a predicate $p$ as a temporal logic formula, to ask whether $p$ is true of a sequence $\sigma$ is to ask whether it is true of the first element of $\sigma$. $\text{Next } g$ is true if $g$ is true at the next time step — a sequence satisfies $\text{Next } g$ if the sequence obtained by removing the first element of the sequence satisfies $g$. A sequence satisfies $g \text{ Until } h$ if $g$ is true until $h$ becomes true — if there is a $k$ such that the first $k - 1$ suffixes of the sequence satisfy $g$ and the $k$-th suffix satisfies $h$. In the special case of $g$ and $h$ not containing temporal operators, this boils down to saying that $g$ is true of the first $k - 1$ states in the sequence, and $h$ is true of the $k$-th state.

Notation: Let $\sigma = s_0s_1s_2 \ldots$ be a sequence in $S$. The following notation is used: $s_i = s_i$ and $\sigma_{\geq i} = s_i s_{i+1} \ldots$

Definition 2. Semantics of TL

1. If $g \in G$ then $\text{Sat}(\sigma, g) = \{s_0\}$
2. $\text{Sat}(\sigma, \neg g) = \neg \text{Sat}(\sigma, g)$
3. $\text{Sat}(\sigma, g \land h) = \text{Sat}(\sigma, g) \land \text{Sat}(\sigma, h)$
4. $\text{Sat}(\sigma, \text{Next } g) = \text{Sat}(\sigma_{\geq 1}, g)$
5. $\text{Sat}(\sigma, g \text{ Until } h) = \bigvee_{i=0}^{\infty}(\text{Sat}(\sigma_{\geq 0}, g) \land \ldots \land \text{Sat}(\sigma_{\geq i-1}, g) \land \text{Sat}(\sigma_{\geq i}, h))$

An advantage of the form of this definition is that it applies to a range of truth domains, not only the standard boolean domain. This paper only presents an algorithm for the fragment of the logic without the until operator. This is an undesirable restriction, made because of the difficulty of the problem. Nevertheless, the version of the problem that we are left with is still difficult and very important so it is a worthwhile goal to explore this restricted problem further.

2.2.2 Examples and Derived Operators

Besides disjunction, other derived operators are possible. The next operator can be nested:

$\text{Next }^k g = \begin{cases} g, & k = 0 \\ \text{Next }^{k-1} g, & k > 0 \end{cases}$

Conceptually, we think of time 0 as being 'now'. The most important derived operator is the $\text{During }$ operator defined as:

$\text{During }([f_0, t_0], \ldots, [f_n, t_n]) g \overset{\text{def}}{=} \bigwedge_{j=0}^{n-1} \bigvee_{k=j}^{t_j} \text{Next }^k g$. 

SART / SACJ, No 24, 1999
Informally, this formula asks whether from time \( f_0 \) through to \( f_1 \) through \( t_1 \), \ldots, and from \( f_n \) through \( t_n \), that \( g \) is true.

Example 2.1 (Example TL formulas). Here are some examples.

1. \([\text{Clk}] = H \land [\text{Reset}] = L \iff [\text{Score}] < 16: \text{At time 0 (now), is the Clock is high, the reset line low, and Score is not greater than 16?}
2. \([\text{Clk}] = H \land [\text{Next}^{10}[\text{Clk}] = L] \land [\text{Reset}] = L]: \text{Is the clock high at time 0, and at time 10 are the clock and reset lines low?}
3. During\([0, 9), (20, 29)\] \([\text{Clk}] = L \land DURING \([10, 19), (30, 39)\] \([\text{Clk}] = H \land DURING \([0, 2]\) \([\text{Reset}] = H \land DURING \([3, 39]\) \([\text{Reset}] = L].

The formula asks if the clock is low for 10ns, then high for 10ns, then low for 10ns, and then high for 10ns; and whether the reset line is high for 3 ns (time 0 through 2 inclusive), and then low from time 3 to time 39.

2.2.3 Symbolic version of the logic

The version of the logic just shown is the scalar version since it contains no variables. Adding variables to the logic enables the logic to become far more expressive in a practical sense. For example, with variables, one can write expressions of the form:

\[
\text{During}\([0, 10]\) \text{Sel} = x \land \text{Ainp} + \text{Binp} = z \]

where Sel is a boolean valued node in a circuit, Ainp and Binp denote collections of nodes which can be considered as having integer values, \( x \) is a boolean valued variable and \( z \) is an integer-valued variable. Nodes are convenient ways to refer to state-holding components in the circuit, whereas variables are ways of concisely encoding a number of formulas in one using implicit universal quantification. In all formulas in this paper, all references to nodes have brackets [. . .] around them; other identifier names (other than the logical operators) are variables. For example the assertion:

\[
\text{Ainp} = x \land \text{Binp} = y \implies \text{Next}^5[C] = x + y
\]

asks of a circuit under study whether for all values of \( x \) and \( y \), if at time 0 Ainp and Binp have the values \( x \) and \( y \) respectively, then at time 5 \( C \) has the value \( x + y \).

The syntax of the symbolic logic is an extension of that of the scalar logic: in addition to scalar formulas we also allow symbolic variables from a set of variables, \( V \).

Definition 3 (The Symbolic Logic — TL). The set of symbolic TL formulas is defined by:

\[
\text{TL} := V \mid G \mid \text{TL} \land \text{TL} \mid \neg \text{TL} \mid \text{Next} \text{TL} \mid \text{Until} \text{TL}
\]

Interpretations: An interpretation of variables is a function mapping from the variables to the boolean values (the example above appears to indicate that we have integer-valued variables — these in fact are not primitive, but formulas involving integer and other valued variables can be encoded using boolean variables). All the examples shown in this paper come from hardware verification where the size of integers is bounded, and hence the encoding into boolean variables straightforward.

Let \( \Phi = \{ \phi : V \to \{f, t\} \} \) be the set of all interpretations. Given an interpretation \( \phi \) of the variables, there is a natural, inductively defined interpretation of TL formulas based on their structure.

Definition 4. If \( \phi : V \to \{f, t\} \), we can extend it to map from TL to TL by:

1. \( \phi(g) := g \) if \( g \in G \).
2. \( \phi(\neg g) := \neg\phi(g) \).
3. \( \phi(g_1 \land g_2) := \phi(g_1) \land \phi(g_2) \).
4. \( \phi(\text{Next} g) := \text{Next} \phi(g) \).
5. \( \phi(\text{Until} g) := \phi(g) \text{Until} \phi(g) \)

(Note that in the above, the case \( \phi(v) \) is not given explicitly since \( \phi : V \to \{f, t\} \)).

Typically in verification the goal is that a formula is true for all interpretations, which is why we informally ask whether a symbolic formula is true of a sequence or set of sequences.

In the introduction, the symbol \( \implies \) was used informally to denote implication. Here it is formally defined.

Definition 5. If \( g, h \in \text{TL} \), \( g \implies h \) if for all sequences \( \sigma, \) and for all interpretations, \( \phi, \) whenever \( \text{Sat}(\sigma, \phi(v)) \) is true then so is \( \text{Sat}(\sigma, \phi(h)) \).

Informally put: \( g \implies h \) iff whenever \( g \) is true of a sequence, \( h \) is true too.

2.3 Modifications

Time-shifting and substitution are defined below. Substitution allows the substitution of symbolic expressions for variables. Time-shifting allows the changing of time to which the formula refers.

Substitution

Let \( F(V) \) be the symbolic expressions over \( V \) (informally TL formulas that do not contain any temporal operators or references to state-holding components). A substitution enables us to replace a variable with a symbolic expression, formally it is a mapping \( \xi : V \to F(V) \). For example, the following is one such mapping: \( \xi_1 \equiv \{ x \mapsto 3, y \mapsto H, z \mapsto a + 3 \times b \} \).

Interpretations are those substitutions whose range contains no variable. Just as interpretations can be extended to map from formulas to formulas (see Definition 4), so can substitutions. The details are omitted, but essentially a substitution can be thought of as a syntactic substitution of expressions for variables throughout a formula.

Example 2.2. Consider:

\[
g \equiv \text{During}[0, 10]\) \([\text{Clk}] = y \land \text{Out} = x < z) \]
then
\[
\xi_1(g) = \text{During}[0, 10]\) \([\text{Clk}] = H \land \text{Out} = 3 < a + 3 \times b) \).
Time-shift:

This is just nesting a formula within the Next operator. If \( g \) is a formula, then 'g time-shifted by n' is just \( \text{Next}^n g \).

Only non-negative time-shifts are allowed.

2.4 Formal statement of problem

The problem this paper addresses can formally be stated as:

Let \( g \) and \( h \) be TL formulas. Do there exist a \( \xi \) and an integer \( n \geq 0 \) such that

\[ g \implies \text{Next}^n \xi(h) \]

If so, what are the \( \xi \) and \( n \) that make the implication true?

There may of course be many \( \xi, n \) pairs that meet this requirement. Generally we want the substitution with the smallest corresponding time-shift.

Duration of match:

If there is a specialisation/time-shift pair \( \xi, n \) (so that \( g \implies \text{Next}^n \xi(h) \)) then it is often the case that \( g \implies \text{Next}^{n+i} \xi(h) \) for many successive \( i > 0 \). Knowing the start and end points of a matching interval is essential to avoid repeated work, so we can generalise the problem to:

Let \( g \) and \( h \) be TL formulas. Do there exist a \( \xi \) and an \( n, d \geq 0 \) such that \( g \implies \text{Next}^{n+i} \xi(h) \) for all \( i \in \{0, \ldots, d\} \). If so, what are the \( \xi \), \( n \) and \( d \) that make the implication true?

Notation 2.1. As a shorthand, the notation \( g \xrightarrow{\xi} h \) stands for the question whether there exists such \( \xi, n, d \) so that the implication holds.

Before looking at algorithms, we first simplify the problem, and then use this simplification to look at different views of the problem. This insight helps us understand how to develop an algorithm.

2.5 Alternative view I

Without the infinite until operator it is possible to write a formula \( \phi \) in a normalised-form (but see the comment on restrictions below): \( \Lambda_{j=0}^{r} \text{Next}^j \phi_j \), where the \( \phi_j \) do not contain any temporal operators and \( r \) is some constant integer.

Given \( g \defeq \Lambda_{j=0}^{r} \text{Next}^j g_j \), \( h \defeq \Lambda_{k=0}^{s} \text{Next}^k h_k \), to know whether there exist a \( \xi, n \) such that \( g \implies \text{Next}^n \xi(h) \) is the same as asking whether there exists a \( \xi, n \) such that \( g_{n+k} \implies \xi(h_k) \), \( 0 \leq k \leq s \). In typical examples \( r \) would be significantly larger than \( s \). In this formulation, we can think of \( g \) and \( h \) as being strings of formulas,

\[ g': g_0 g_1 g_2 \ldots \ldots \ldots \ldots g_{r-1} g_r \]
\[ h: h_0 h_1 \ldots \ldots \ldots h_s \]

where we want to match \( h \) in \( g \) - a generalised string matching problem, or a unification where we need to unify times as well as values.

2.6 Alternative view II

This formulation gives a useful insight into the problem, but the difficulty with it is that although \( r \) and \( s \) are large (typical values might be \( r = 10^5 \), \( s = 10^6 \)), there are likely to be very long sub-strings (of length several thousand) which have repeated 'symbols' - i.e. \( g_i = g_{i+1} \) for very many successive \( i \). The reason for this is that in practice the most important operator is the bounded always operator \( \text{During}[] \). This means that the normalised form is usually better given as:

\[ g \defeq \bigwedge_{j=0}^{e} \text{During}[(u_j, v_j)] g_j \]
\[ h \defeq \bigwedge_{k=0}^{e'} \text{During}[(w_k, x_k)] h_k \]

Now the subformulas have intervals associated with them. Our problem is to try to 'slide' the intervals associated with \( h \) forward until all the intervals of \( h \) are 'covered' by intervals of \( g \) and the implication holds between the corresponding \( g_j \) and \( h_k \). One of the complexities in this is that it might take several intervals of \( g \) to cover one of \( h \)'s, or one interval of \( g \) might cover several of \( h \)'s, or both.

But although there is this complexity with this representation, it is very compact, and so is the one used in the rest of this paper and from now on it is assumed that formulas are written in this way.

Restriction on the logic The normal form requires a little thought in dealing with disjunction. The normal form works for formulas like \( \text{Next} (g_1 \lor g_2) \) if the \( g_i \) don't contain temporal operators. However, the normal form presented above is not general enough to deal with formulas where disjunction is used to connect sub-formulas referring to different instants in time (e.g. \( g_1 \lor \text{Next} g_2 \)). In this case, a more general normal form can be used, representing a formula by a list of lists. This has been implemented in the VossProver, but has not been used since it has not yet turned out not to be useful in practice since no real examples have presented this case. The rest of this discussion therefore assumes that the simple normal form above can be used. The algorithm can in principle be generalised but the complexity of the algorithm has not been investigated.

2.7 Examples

Some simple example problems are shown below.

1. \( g \defeq \text{During}([100, 115], [130, 150]) [A] = f \]
\( h \defeq \text{During}([10, 20], [40, 50]) [A] = x \geq 10 \)
\( \xi = \{ f \rightarrow x \geq 10 \}; \text{shift}=90; \text{slack}=5 \). Any shift between 90 and 95 will work.

2. \( g \defeq \text{During}([100, 104]) [\text{clk}] = H \)
\( \text{During}([105, 115], [120, 135]) [\text{clk}] = H \)
\( h \defeq \text{During}([0, 10], [20, 30]) [\text{clk}] = H \}
\( \xi = \{ \}; \text{shift}=100, \text{slack}=5 \)
3. A first algorithm

This section presents an algorithm to solve the problem. This section is structured as follows. First, data representation is presented, followed by a description of how implication between non-temporal logical formulas is tested. This is used as the basis for the generalised unification which is presented in two steps: first unification between non-temporal formulas, and then unification between TL formulas.

3.1 Representation of formulas

All formulas are represented as abstract data types. At the top level are TL formulas, which can be predicates or composed from simpler TL formulas using temporal operators or the usual logical operators like conjunction and negation.

A predicate is a boolean expression (usually evaluated with respect to the state of a circuit), which could just be boolean data of some sort, or a relation involving integer expressions or vectors. Typical relations are equality and comparison. The integer expressions can include constants, variables, addition, subtraction, multiplication, base-2 exponentiation, and the taking of bit-slices. Bit-slice operations are important in the application domain: if one represents an integer expression in binary (i.e. as a bit-vector), taking a bit slice involves extracting out a segment of the bit-vector and then considering that as an integer expression. Both signed and unsigned arithmetic is supported.

We now look at how reasoning about predicates and data is done.

Normalisation routines: A set of normalisation routines have been built up that attempt to put predicates into a normal form. This involves term rewriting using criteria such as: a lexicographic ordering defined on terms; removing redundant operations (e.g. multiplying by 1); finding common factors; expanding simple cases (e.g. $2^0 = 1$); and reducing constant expressions wherever possible. An example of the type of reduction that gets done is:

$$x[n..0] + 2^{n+1} \times x[n+1] \longrightarrow x[n+1..0],$$

where $x[j..i]$ is that slice of the expression $x$ starting at bit $i$ (least significant) going to bit $j$ (most significant).

Binary Decision Diagrams

Binary decision diagrams (BDDs) are a canonical data representation method for boolean data [1]. They are a very compact representation and can be used to perform a variety of manipulations efficiently. FL can represent boolean data as BDDs, and routines have been written that can convert bounded integer expressions into vectors of BDDs. Boolean, arithmetic and relational operations can be implemented as BDD operations. Therefore, BDDs are a useful component for reasoning, but in this context have many limitations.

User supplied information

It is also possible for the user to provide 'domain information'. A hook is provided for expressing more complicated symbolic expressions and the user can provide facts about such expressions.

3.2 Testing implication between logical formulas

The first issue is how to test implication between two logical formulas that do not contain any temporal operators. As shown above, the types of expression allowed are very general, and include multiplication and restricted exponentiation. Although an efficient general algorithm to test implication is not possible, practical experience indicates that good heuristics are able to solve many (but not all) problems in practice. (However, there are difficult issues in tool development and user interaction with tools which have not yet been solved.)

The algorithm for testing whether $g \implies h$ first normalises $g$ and $h$ (using the techniques described above) and then applies rules such as the following to determine implication:

$$u \implies t \quad f \implies v$$
$$t \implies u, \text{ iff } v \equiv t \quad u \implies v \text{ if } u \equiv v$$
$$u' \land v \implies u, \text{ if } u' \implies u$$
$$v \lor (c_i \land u_i) \implies v_i, \text{ if } c_i \land u_i \implies v_i \lor c_i = t$$
$$(u = v) \implies (x = y), \text{ if } \{u, v\} \equiv \{x, y\}$$

Wherever possible commutativity and associativity are applied in conjunction with the above rules, which are applied recursively on the structure of the formulas. If the algorithm succeeds then we know that $g \implies h$. If the algorithm does not succeed we cannot deduce that $g \not\implies h$ since the proof and normalisation system is not complete. The tool takes care never to use a negative answer so as to imply that the result is false.

3.3 Unification between logical formulas

The previous section looked at how implication between two formulas is tested. Here we look at how, given two formulas $g$ and $h$ not containing temporal operators we can determine whether $g \triangleleft \equiv h$. The algorithm works on the syntactic structure of formulas in a similar fashion to the implication-testing algorithm presented in the previous section.
3.3.1 Unifying predicates

The difference arises when we get to the 'base' case when we have two predicates $p_g$ and $p_h$ (recall that predicates contain no boolean connectives or temporal operators) and we wish to know whether $\leftrightarrow p_h$. Predicates contain expressions made up of relational and other operators, constants, variables and references to the state space.

(1) Let's take a concrete example: $[C] + [D] = x + 2 \times y$. $[C]$ and $[D]$ are references to the state space (in the application domain, they are nodes in a circuit being verified); $x$ and $y$ are variables. To evaluate the truth of a predicate with respect to a state, take the values of the nodes $[C]$ and $[D]$ and plug them into the expression. This gives the set of interpretations of $x$ and $y$ for which the formula is true.

The algorithm first checks whether the relations to the state space in $p_g$ and $p_h$ are the same. If $p_h$ has references to components that $p_g$ does not have then the algorithm decides that the relationship doesn’t hold. Sometimes this will mean that valid relationships are not found as in: $[D] - [D] + [C] < x \iff [C] < x$. But this test is relatively cheap to perform and very rarely gives a false negative in practice (as usually normalisation routines eliminate expressions like $[D] - ([D])$).

The second step applies a simple matching routine. Suppose the predicates have the structure like this: $p_g \equiv p_h \Rightarrow p_h \equiv p_h$, where $\Rightarrow$ is a relational operation. If $p_g \equiv p_h$, then the substitution $p_h \mapsto p_h$ is a potential substitution that could be applied. If this step fails we can also use commutativity (depending on what $\Rightarrow$ is), e.g., see if $p_g \equiv p_h$.

For example, with $[C] + [D] = x + 2 \times y \iff [C] + [D] = x$ we would get as a match $x \mapsto x + 2 \times y$.

Similar simple algorithms are applied to other types of predicates (the one above is the most complicated case). If the two predicates have different structures, then it is highly unlikely that a match will be found, unless the predicate on the right is just a boolean variable, in which case the potential substitution is just $p_h \mapsto p_h$.

The substitutions are only potential for several reasons. First, this match is likely to be just one of many from a decomposition of more complicated formulas. Other matches that are found may be inconsistent with this one. Second, in a substitution like $p_h \mapsto p_h$, $p_h$ may be an expression rather than a variable (in practice usually it is a variable). This may or may not be a problem. Third, such a substitution may be illegal (like $1 \mapsto 2$).

3.3.2 Collecting all potential matches

This algorithm for finding a unification between two predicates is used to test unification between two non-temporal logical formulas, using the syntactic structure of the formulas in a similar way to the algorithm that tests implication described in Section 3.2. The difference is that (1) a test between sub-formulas succeeds either if implication holds or if there is a potential substitution that unifies them, and (2) a 'yes' answer comes with a list of potential substitutions that are returned to the caller.

The rest of the paper assumes the existence of a procedure $\text{Match}(g, h)$ (for $g$, $h$ which are non-temporal formulas) which implements the algorithm just presented: it either returns $f$ (meaning that $g \not\leftrightarrow h$ is false) or $t$ (meaning that $g \leftrightarrow h$ might be true) together with the list of potential substitutions.

3.4 Unifying two temporal logic formulas

Overview: The two formulas $g$ and $h$ to be unified are normalised so that they are represented as lists of start-time, end-time, logical formula tuples $g'$ and $h'$. For example, the last formula in Example 2.1 would be represented by the following list:

$$[(0, 2, [\text{Reset}] = H \land [\text{Clk}] = L), (3, 9, [\text{Reset}] = L \land [\text{Clk}] = L), (10, 19, [\text{Reset}] = L \land [\text{Clk}] = H), (20, 29, [\text{Reset}] = L \land [\text{Clk}] = L), (30, 39, [\text{Reset}] = L \land [\text{Clk}] = H)]$$

We then 'slide' the $h'$ list over the $g'$ list (treating the times in the $h'$ list as being relative to the yet unknown shift value, and the times in the $g'$ list as absolute). This process is effectively a string matching algorithm, though the use of intervals and logical formulas instead of 'letters' complicates the algorithm considerably. The algorithm uses a naive approach to this matching problem, and the question of how more intelligent algorithms could be used (such as those presented in [2, 5]) needs attention.

As the matching process continues, the list of potential substitutions and time-shifts is kept.

Detail: The algorithm was implemented in a functional language. The code segments shown below are given at an abstract level and presented in an imperative style. The presentation is somewhat simplified and there are some special cases that must be dealt with.

The main loop of the unification algorithm is shown in Figure 1. First, the two formulas are normalised into a list (of normalised start-time, end-time, logical formula tuples, called $g'$ and $h'$). Then the code enters a loop. The first step is to find the first tuple in the list $g'$ that potentially matches the first tuple in $h'$ (performed by $\text{FindFirst}$ described below). If a match is found, the shift (i.e., the time that $h$ would have to be moved forwards for the match to happen), the slack (i.e., any shift between shift and $\text{shift} + \text{slack} - 1$ will work) and a potential substitution are returned. In addition, $g'$ and $h'$ are modified to chop off the non-matching part of the prefix of $g'$, and those parts of $g'$ and $h'$ that are covered by the first match. If no first match is found, then the loop terminates.

After the first match has been found the modified $g'$ and $h'$ must be matched. In performing this match, it is possible to increase shift by up to slack. If the rest of the lists cannot be matched, then the algorithm tries to find a new first match (i.e., the next tuple in the $g'$ list that matches...
function TLMatch(g, h) : (Q, sublist);  
g' ← normalise(g);  
h' ← normalise(h);  
loop  
  (found, shift, slack, subs, g', h') ← FindFirst(g', h');  
  if not found then return false ;  
  (found, shift, slack, subs) ← MatchRest(start, shift, slack, g', h');  
  subs ← subs ∪ subs1 ;  
  if found then  
    return (true, (subs, start, slack));  
  end;  
end;

The algorithm seen so far returns a proposed shift and substitution. However, as has been pointed out, the heuristics that produce the substitutions may produce invalid substitutions. The final phase of the algorithm, therefore, is to perform the substitution and time-shift on h and then to check that \( g \Rightarrow h \).

4 Examples

The algorithm presented in the previous section was implemented in the VossProver in FL, a lazy functional language [8] similar to ML. FL can be used as both a systems programming language and as the interface to the VossProver verification system. The Voss system contains an efficient implementation of BDDs and FL represents booleans internally as BDDs.

For the simple examples shown in Section 2.7, the time the algorithm takes is too small to be determined accurately.

The algorithm was then used in practice on verifying two large circuits in the IFIP WG10.5 Hardware Benchmark Verification Suite [6, 9], Benchmarks 17 (a multiplier) and 21 (a systolic filter). The verification of these benchmarks is described in [3]. Alternative approaches can be found in [7]. The advantage of [3] over previous work [4] is that although the cost of doing unification adds to the computational cost of verification, significant savings in human interaction are achieved.

In a run of verifying Benchmark 21, the unification algorithm presented here is called several hundred times, and so the cost of performing the unification becomes significant — roughly 30% of the overall cost of the computation.

The types of formulas that are involved in the computation are very long and boring, which makes them inappropriate to display here — and very difficult for a human to reason about. To give a flavour of examples occurring in practice, the appendix gives an extract from a real example.

Measures of complexity of the formulas are the length of the formula when represented in normalised form, and the longest conjoint of one of the sub-formulas. In the case studies done, common cases are: length of the formula \( h \) of 6, longest conjoint in \( h \) 5; length of the formula \( g \) of

Figure 1: Code for matching temporal logic formulas

in the first one in the \( h' \) list).

The FindFirst function (also shown in Figure 1) works by scanning the \( g' \) list until it finds an element that matches the first interval in \( h' \). It then checks to see whether any subsequent, contiguous elements of \( g' \) also match \( h_0 \). With this it can compute the initial shift and slack values, and the initial substitutions. Note that the match may fail even if it does find an element of \( g' \) that matches \( h_0 \) if the interval that the element of \( g' \) covers is not big enough to cover the interval \( v_0 \ldots v_0 \). Finally, to make subsequent steps in the algorithm more efficient, the prefixes of the \( g' \) and \( h' \) list already considered are chopped off by the suffix procedure.

Given initial shift and slack values, the MatchRest procedure shown in Figure 2 methodically works through the \( g' \) and \( h' \) lists to check that from the initial shift value, each element of \( h' \) is covered by a corresponding element in \( g' \). This requires not only checking that each \( h_i \) is matched by an appropriate \( g_j \) but that the time intervals also match up. There are a few issues that must be checked. First, it might not be a problem if \( h_0 \) does not match with \( g_0 \) or even the first few \( g_j \). We can adjust the shift value as much as the slack value allows to skip over a few non-matching \( g_j \). In practice, this happens quite often.

Second, there are two reasons we may reduce slack: (1) if we update the shift (as described above), and (2) if the slack given by the current interval is less than the slack previously computed.

Third, once we know that \((w_j, x_j, g_j)\) matches \((u_i, v_i, h_i)\), we need to continue doing the checking for subsequent intervals. If \( x_j - w_j = (v_i - u_i) \), this is straightforward. If not, we must make sure that we either check that the part of the \( i \)-tuple not covered by the \( j \)-tuple is still checked, or that the part of the \( j \)-tuple not needed to cover the \( i \)-tuple can still be used in the matching.

Validating the result

The algorithm seen so far returns a proposed shift and substitution. However, as has been pointed out, the heuristics that produce the substitutions may produce invalid substitutions. The final phase of the algorithm, therefore, is to perform the substitution and time-shift on \( h \) and then to check that \( g \Rightarrow h \).

4 Examples

The algorithm presented in the previous section was implemented in the VossProver in FL, a lazy functional language [8] similar to ML. FL can be used as both a systems programming language and as the interface to the VossProver verification system. The Voss system contains an efficient implementation of BDDs and FL represents booleans internally as BDDs.

For the simple examples shown in Section 2.7, the time the algorithm takes is too small to be determined accurately.

The algorithm was then used in practice on verifying two large circuits in the IFIP WG10.5 Hardware Benchmark Verification Suite [6, 9], Benchmarks 17 (a multiplier) and 21 (a systolic filter). The verification of these benchmarks is described in [3]. Alternative approaches can be found in [7]. The advantage of [3] over previous work [4] is that although the cost of doing unification adds to the computational cost of verification, significant savings in human interaction are achieved.

In a run of verifying Benchmark 21, the unification algorithm presented here is called several hundred times, and so the cost of performing the unification becomes significant — roughly 30% of the overall cost of the computation.

The types of formulas that are involved in the computation are very long and boring, which makes them inappropriate to display here — and very difficult for a human to reason about. To give a flavour of examples occurring in practice, the appendix gives an extract from a real example.

Measures of complexity of the formulas are the length of the formula when represented in normalised form, and the longest conjoint of one of the sub-formulas. In the case studies done, common cases are: length of the formula \( h \) of 6, longest conjoint in \( h \) 5; length of the formula \( g \) of
function MatchRest(shift, slack, subst, \[(w_0, x_0, g_0), \ldots, (w_s, x_s, g_s)\], \[(u_0, v_0, h_0), \ldots, (u_r, v_r, h_r)\])
joint = 0; // used for the g's
i = 0; // used for the h's
subs = []; // this
while i < r ∧ j < s do
  k = j; // jump on non-matches
  while ¬(Match(g_j, h_i)) ∧ j < s ∧ slack > 0 do
    j = j + 1;
  end;
  if j > s then return false;
  // Update shift, slack, subst
  δ = w_j - w_k;
  shift = shift + δ;
  slack = slack - δ;
  subst = subst ∪ subst);
  tocover = v_0 - u_0;
  if slack < 0 then return false;
  while Match(g_{j+1}, h_i) ∧ x_j + 1
                            \[= w_{x+1} ∧ z \leq s do
                                          \]
    z = z + 1;
  end;
  tslack = x_j - w_j - tocover;
  slack = min{tslack, slack};
  // Now check what has been covered
  if tocover > cover then
    j = j + 1;
    u_i = u_i + cover + 1;
  elseif cover > tocover then
    i = i + 1;
    w_j = w_j + tocover + 1;
  else
    i = i + 1;
    j = j + 1;
  end;
end;
return \[i > r, shift, slack, subs\];
end;

Figure 2: The MatchRest function

80, longest conjoint in \(g\) of 16. In our example, the \(g\) is a formula which describes a known result of an execution of a circuit; \(h\) is an antecedent of a rule of inference; and if we are able to match \(g\) and \(h\) then we can infer something new about the circuit's execution.

Analysis: In the worst case, the cost of the string matching part of the algorithm is \(nm\) where \(n\) is the length of the normalised representation of \(g\) and \(m\) that of \(h\). But the cost is also heavily influenced by the cost of unification between non-temporal formulas, for which a theoretical analysis is very difficult, especially for the average case complexity which is most important. Moreover, the cost of normalising formulas has to be taken into account.

Experimental analysis is complicated by two factors of the VossProver: garbage collection and caching. Since garbage collection can happen at any time, the cost is highly variable. Secondly, FL can cache results of function calls which means that if the function is called again with the same arguments, the result does not have to be recomputed. Many of the normalisation routines can benefit significantly from caching. This means that the cost of performing unification depends on operations that have been previously performed.

However, some rough estimates are available. For example on a Sun Ultra 4, the cost of a unification on formulas of the complexity mentioned above is approximately 1.5s (this includes the cost of normalisation). It should be borne in mind that FL is an interpreted functional language, and that its range of data structures is not always the best for fast execution (for example, scanning a list is much more expensive than scanning an array).

The other criterion for analysing the algorithm is its coverage. Since it is a heuristic, it does not guarantee to find all unifications. (For example, the algorithm depends heavily on how good the normalisation algorithms are.) As can be seen a range of fairly complex formulas have been tested and the algorithm succeeded on all. However, the application domain is relatively narrow one.

5 Conclusion

The problem of unifying two temporal logic formulas poses interesting theoretical and practical questions, and has very important practical applications. This paper has introduced the problem and presented a simple algorithm that solves a restricted version of the problem. Although naïve, the algorithm has proved to be robust with an acceptable computational cost. However, there is a need to improve both the coverage and the performance of the algorithm.

There are a number of areas that need research. Integrating new domain knowledge. To be effective the algorithm needs to have a powerful means of reasoning about data. After some years of development, the VossProver has repertoire of routines that enable it to reason about a range of arithmetic problems that arise in hard-
ware verification problems. However, when faced with a new problem it is necessary for the human user to provide new domain knowledge. At the moment, this is handled by extending the VossProver directly but this is not a satisfactory solution. This is an area that needs work.

Improving performance. Exploring how some of the standard string-matching and unification algorithms can be generalised to deal with this problem appears to be a profitable line to follow.

From a practical point of view, significant performance gain can be made by implementing the code in C as part of the Voss system. Not only would compiled code run much faster than interpreted code, but FL's limited data structures would probably not be sufficient for more sophisticated matching algorithms.

Extending the theory to deal with the full TL. While the omission of the infinite until operator still leaves a logic rich enough to express very many problems, it would be very desirable to be able to include this operator. The most obvious difficulty here is determining appropriate normal forms.

Acknowledgements: This research was supported by the National Research Foundation under GUNS 2039523 and 2042569 and the University of the Witwatersrand Research Committee. The paper was written while I was on sabbatical at the Laboratoire D'Analyse et d'Architecture des Systèmes, Centre National de la Recherche Scientifique, Toulouse. Vashti Galpin and anonymous referees gave valuable comments on a draft of this paper.

References


A Example TL Formula

Here is an example of a TL formula in normalised form: less than 5% of a formula that arose in the verification of a benchmark. When represented in normalised form, there are 83 start - finish - formula tuples. The most complex of the formulas is a conjunction of 16 predicates.

$$\begin{align*}
g &\triangleq \left(1.3, \text{StoreRes} = F \text{ and StoreWgt} = \text{Final} \text{ and StoreStr} = T \text{ and SelectWgtStr} = T \text{ and SelectWgt} = T \text{ and SelectWgtStr} = T \text{ and StreamIn.0} = w[0][3-0] \text{ and CLK1} = F \text{ and CLK2} = T \text{ and Reset.N} = T, \right) \\
&\text{(4,9,StoreRes} = F \text{ and StoreWgt} = F \text{ and StoreStr} = T \text{ and SelectWgtStr} = T \text{ and SelectWgt} = T \text{ and SelectWgtStr} = T \text{ and StreamIn.0} = w[0][3-0] \text{ and CLK1} = F \text{ and CLK2} = T \text{ and Reset.N} = T), \ldots \text{ lots left out here ... (34500,34999,}
\end{align*}$$

$$\begin{align*}
\text{StoreRes} = T \text{ and StoreWgt} = F \text{ and StoreStr} = T \text{ and SelectWgtStr} = T \text{ and CLK1} = F \text{ and CLK2} = T \text{ and Reset.N} = T \text{ and StreamIn.3} = d[5][3-0] \text{ and Dpath2.0} = w[3][3-0] \text{ and Dpath2.1} = w[2][3-0] \text{ and Dpath2.2} = w[1][3-0] \text{ and Dpath2.3} = w[0][3-0] \text{ and ResultIn.1} = d[4][3-0] \text{ * w[0][3-0] + d[5][3-0] * w[1][3-0]}, \\
(35000,35000,\text{StoreRes} = T \text{ and StoreWgt} = F \text{ and StoreStr} = T \text{ and SelectWgtStr} = T \text{ and CLK1} = T \text{ and CLK2} = F \text{ and Reset.N} = T \text{ and StreamIn.3} = d[5][3-0] \text{ and Dpath2.0} = w[3][3-0] \text{ and Dpath2.1} = w[2][3-0] \text{ and Dpath2.2} = w[1][3-0] \text{ and Dpath2.3} = w[0][3-0] \text{ and ResultIn.1} = d[4][3-0] \text{ * w[0][3-0] + d[5][3-0] * w[1][3-0]}), \ldots
\end{align*}$$
The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications of Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript
Manuscripts for review should be prepared according to the following guidelines:

- Use wide margins and 1½ or double spacing.
- The first page should include:
  - the title (as brief as possible)
  - the author's initials and surname
  - the author's affiliation and address
  - an abstract of less than 200 words
  - an appropriate keyword list
  - a list of relevant Computing Review Categories
  - Tables and figures should be numbered and titled.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text according to the Harvard method.

Manuscripts accepted for publication should comply with guidelines as set out on the SACJ web page, http://www.cs.up.ac.za/sacj which gives a number of examples.

SACJ is produced using the \LaTeX\ document preparation system, in particular \LaTeX\ 2e. Previous versions were produced using a style file for a much older version of \LaTeX, which is no longer supported.

Please see the website for further information on how to produce manuscripts which have been accepted for publication.

Authors of accepted publications will be required to sign a copyright transfer form.

Charges
Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30.00 per final page for contributions which require no further attention. The maximum is R120.00, prices inclusive of VAT.

These charges may be waived upon request of the author and the discretion of the editor.

Proofs
Proofs of accepted papers may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications
Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words. Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book Reviews
Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement
Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialised production costs, which will be borne by the advertiser. Enquiries should be directed to the editor.