COUNCIL OF THE
EUROPEAN ASSOCIATION FOR
THEORETICAL COMPUTER SCIENCE

PRESIDENT: Burkhard Monien Germany
VICE PRESIDENTS: Giorgio Ausiello Italy
Don Sannella United Kingdom
Paul Spirakis Greece
Treasurer: Dirk Janssens Belgium
Bulletin Editor: María Serna Spain

Luca Aceto Iceland
Susan Albers Germany
Krzysztof Apt The Netherlands
Pierre-Louis Curien France
Josep Díaz Spain
Zoltán Ésik Hungary
Fedor Fomin Norway
Lance Fortnow USA
Leslie Ann Goldberg United Kingdom
Giuseppe F. Italiano Italy
Christos Kaklamanis Greece
Juha Karhumäki Finland

Antonín Kucera Czech Republic
Jan van Leeuwen The Netherlands
Madhavan Mukund India
Mogens Nielsen Denmark
Catuscia Palamidessi France
David Peleg Israel
Giuseppe Persiano Italy
Jean-Eric Pin France
Vladimiro Sassone United Kingdom
Andrzej Tarlecki Poland
Thomas Wilke Germany
Gerhard Woeginger The Netherlands

PAST PRESIDENTS:
Maurice Nivat (1972–1977)
Mike Paterson (1977–1979)
Arto Salomaa (1979–1985)
EATCS Council Members

Email Addresses

Luca Aceto ........................................ luca@ru.is
Susanne Albers ................................. albers@informatik.hu-berlin.de
Giorgio Ausiello ................................. ausiello@dis.uniroma1.it
Krzysztof Apt ........................................... apt@cwi.nl
Pierre-Louis Curien ............................. curien@ppps.jussieu.fr
Josep Diaz ......................................... diaz@lsi.upc.es
Zoltán Ésik .......................................... ze@inf.u-szeged.hu
Fedor Fomin ........................................... fomin@ii.uib.no
Lance Fortnow ................................. fortnow@eecs.northwestern.edu
Leslie Ann Goldberg ........................... L.A.Goldberg@liverpool.ac.uk
Giuseppe F. Italiano ......................... italiano@disp.uniroma2.it
Dirk Janssens ................................. Dirk.Janssens@ua.ac.be
Christos Kaklamanis ......................... kakl@ceid.upatras.gr
Juhani Karhumäki ................................. karhumak@cs.utu.fi
Antonín Kucera ....................................... tony@fi.muni.cz
Jan van Leeuwen ................................. jan@cs.uu.nl
Burkhard Monien ................................... bm@upb.de
Madhavan Mukund ............................... madhavan@cmi.ac.in
Mogens Nielsen ..................................... mn@brics.dk
Catuscia Palamidessi ......................... catuscia@lix.polytechnique.fr
David Peleg ......................................... peleg@wisdom.weizmann.ac.il
Giuseppe Persiano ............................... giuper@dia.unisa.it
Jean-Eric Pin ....................................... Jean-Eric.Pin@liafa.jussieu.fr
Don Sannella ....................................... dts@dcs.ed.ac.uk
Vladimiro Sassone ............................... vs@ecs.soton.ac.uk
Maria Serna ......................................... mjserna@lsi.upc.edu
Paul Spirakis ........................................ spirakis@cti.gr
Andrzej Tarlecki ................................. tarlecki@mimuw.edu.pl
Thomas Wilke ...................................... wilke@ti.informatik.uni-kiel.de
Gerhard Woeginger .............................. gwoegi@win.tue.nl
The bulletin is entirely typeset by \texttt{pdftex} and \texttt{context} \texttt{txfonts}. The Editor is grateful to Ivan Couto for his support.

All contributions are to be submitted electronically through the Bulletin’s web site and must be prepared in \LaTeX\ using the class beatcs.cls (a version of the standard \LaTeX\ \texttt{article} class). All sources, including figures, and a reference PDF version must be included in the submission.

Pictures are accepted in EPS, JPG, PNG, TIFF, MOV or, preferably, in PDF. Photographic reports from conferences must be arranged in ZIP files layed out according to the format described at the Bulletin’s web site. Those reports will appear only in the electronic edition of the Bulletin.

We regret we are unfortunately not able to accept submissions in other formats, or indeed submission not \textit{strictly} adhering to the page and font layout set out in beatcs.cls. We shall also not be able to include contributions not typeset at camera-ready quality.

The details can be found at http://www.eatcs.org/bulletin, including class files, their documentation, and guidelines to deal with things such as pictures and overfull boxes. When in doubt, email bulletin@eatcs.org.

Deadlines for submissions of reports are January, May and September 15th, respectively for the February, June and October issues. Editorial decisions about submitted technical contributions will normally be made in 10/15 weeks. Accepted papers will appear in print as soon as possible thereafter.

The Editor welcomes proposals for surveys, tutorials, thematic issues of the Bulletin dedicated to currently hot topics, and letters to the editor, as well as suggestions for new regular sections.

The EATCS home page is http://www.eatcs.org
# Table of Contents

**EATCS Matters**  
Letter from the President ........................................... 3  
Letter from the Editor .................................................. 5  

**Institutional Sponsors** ............................................. 7  

**EATCS News**  
The Japanese Chapter, by R. Uehara ..................................... 11  
News from India, by M. Mukund ........................................ 13  
News from New Zealand, by C.S. Calude ............................... 17  

**The EATCS Columns**  
The Complexity Column, by J. Torán  
On the Notion of Bit Complexity, by C. Diem .......................... 35  
The Concurrency Column, by L. Aceto  
Sessions, from Types to Programming Languages, by V.T. Vasconcelos 53  
The Distributed Computing Column, by P. Fatourou  
Failure Detectors to Solve Asynchronous k-set Agreement: A Glimpse of Recent Results, by M. Raynal ......................... 74  
The Logic in Computer Science Column, by Y. Gurvich  

**The Puzzle Corner**, by L. Rosaz ...................................... 123  

**Reports from Conferences**  
The 19th Workshop on Concurrency, Specification and Programming (CS&P 2010), by M. Kudlek .................................. 127  

**EATCS Leaflet** .......................................................... 129
Letter from the President

Dear EATCS members,

I hope that you all have had an excellent start into the year 2011 and I take the opportunity to wish you well in your time and your work in the theoretical computer science community.

The organization of the next ICALP in Zurich is proceeding well and the corresponding council decisions taken in Bordeaux have been implemented as far as possible. Especially, in Bordeaux, we agreed that the proceedings of ICALP 2011 should again be produced by Springer within the ARCoSS series. However, two innovations need to be highlighted here: firstly, it will no longer be compulsory for ICALP participants to buy printed proceedings. Secondly, EATCS, in cooperation with the MPI in Saarbrücken, provided sponsorship for ten 500-Euro student scholarships. The scholarships will be used to support participation of students in ICALP 2011 by covering early registration and possibly some of the local expenses. The applications will be reviewed by the ICALP 2011 conference and PC chairs. Preference will be given to PhD students from countries where access to funds is limited and who will present papers at the conference.

The organization of ICALP 2012 has already started. ICALP will be organized next year in Warwick by Artur Czumaj and his team. In this context, let me again draw attention to the fact that the centennial of the birth of A. M. Turing is in 2012 and ICALP will be part of the celebration of
this event. The conference itself will have the same tracks as in 2011, and in the meantime we have successfully completed the search for the Program Chairs for ICALP 2012. We are very pleased that we could win our first choice candidates Kurt Mehlhorn (track A), Andrew Pitts (track B) and Roger Wattenhofer (track C) to act as PC chairs of the scientific program of ICALP 2012.

Also, I am very pleased that the EATCS website and especially the new section for young researchers within the members-only area are very well accepted. Our secretarial office regularly updates the information about current Postdoc and other research positions. In this context, I would like to encourage you to continue sending corresponding news to our secretarial office.

Finally, all that remains for me is to encourage you to send your best paper to the next ICALP so that we can keep the standard as one of the world leading conference in theoretical computer science. See you in Zurich!

Burkhard Monien, Paderborn
January 2011
Letter from the Bulletin Editor

Dear Reader,

Welcome to the February 2011 issue of the Bulletin of the EATCS. First of all, I wish all of you a happy and successful 2011.

I have some news to report. Starting from this issue, Ryuhei Uheara will be in charge of informing us on the novelties of the Japanese Chapter of the EATCS. I'm also glad to announce that from the June issue, V. Arvind will be the new editor of The Computational Complexity Column. I want to express my gratitude to Jacobo Torán and Kazuhsa Makino for their great contribution and for the effort devoted to the Bulletin for so many years. My warmest thanks to you.

Regarding the present issue, apart from the interesting Columns contributions, our reporters bring you news on EATCS and TCS activities all over the world. Please, read them carefully!

I trust that you will find a lot of interesting material to read in this issue.

Maria Serna, Barcelona
February 2011
INSTITUTIONAL SPONSORS
BiCi, Bertinoro international Center for informatics
   Bertinoro, Italy

CTI, Computer Technology Institute
   Greece

Elsevier
   Amsterdam, The Netherlands

MADALGO, Center for Massive Data Algorithmics
   Aarhus, Denmark

Microsoft Research
   Cambridge, United Kingdom

Springer-Verlag
   Heidelberg, Germany
REPORT FROM THE JAPANESE CHAPTER

R. Uehara (JAIST)

Kyoto Prize

The Kyoto Prize is held by the non-profit Inamori Foundation, and it is one of the most famous prizes in Japan. According to its web site:

The Kyoto Prize is an international award to honor those who have contributed significantly to the scientific, cultural, and spiritual betterment of mankind. The Prize is presented annually in each of the following three categories: Advanced Technology, Basic Sciences, and Arts and Philosophy.

November 10, 2010, Dr. László Lovász (Eötvös Loránd University) receives the 26th annual Kyoto Prize in Basic Sciences which is the award for his outstanding contributions to the advancement of both the academic and technological possibilities of the mathematical sciences. Congratulations!!

The details of the prize can be found at the web site. Japanese TCS society celebrates it, and the following two workshops were held in Kyoto and Tokyo:

Mathematical Development of Algorithm Science: The following lectures were given on November 12, 2010 at Kyoto International Conference Center

- The Mathematical Challenge of Very Large Networks
- Combinatorial Geometry: Mathematics for Geometric Data Processing
- CompView and the Lovász Local Lemma
- Convexity in Combinatorial Optimization
- Structure Theorems and Decomposition Theorems in Graph Theory

László Lovász
Takeshi Tokuyama
Osamu Watanabe
Satoru Iwata
Ken-ichi Kawarabayashi

You can find the abstracts in English at the PDF file on the web.

Kyoto Prize Satellite Workshop in Tokyo: The following lectures were given on November 16-18, 2010 at Tokyo Institute of Technology

1http://www.inamori-f.or.jp/e_kp_out_out.html
2http://www.kyotoprize.org/news/pressrel/pressrel_040810_lovasz.htm
3http://www.inamori-f.or.jp/laureates/k26_b_laszlo/img/wksab_e.pdf
Cubic and Higher Forms
Ravi Kannan

Optimal Sink-Stable Sets
András Frank

Some Recent Results on the Duality Gap
András Sebö

The VPN Problem and Extensions
Bruce Shepherd

Traveling Salesman Problems
William Cook

Left and Right (a journey through jungle of arrows)
Jaroslav Nešetřil

Hyperbolic Surface Subgroups of One-Ended Doubles of Free Groups
Sang-il Oum

Hard Functions for Low-Degree Polynomials over Prime Fields
Hidetoki Tanaka

Effective Principal Component Analysis
Santosh Vempala

Anatomy of a Young Giant Component in the Random Graph
Jeong Han Kim

The Lovász Local Lemma
Bruce Reed

On the Topology of Graphons
László Lovász

Extendable Structures in Graphs
Michael Plummer

Average Degree Condition Forcing Complete Graph Immersion
Bojan Mohar

A New Proof for the Two Disjoint Odd Cycles Theorem
Kenta Ozeki

The Edge Disjoint Paths Problem in Eulerian Graphs and 4-Edge-Connected Graphs
Yusuke Kobayashi

Tree Metrics and Edge-Disjoint S-paths
Hiroshi Hirai

A Combinatorial Characterization of a Certain Class of 3-dimensional Rigidity Matroids
Shin-ichi Tanigawa

On the Graph Limit Theory
Balázs Szegedy

You can find the details of the workshop at the web site4. At these workshops, many exciting talks were given by gorgeous invited speakers!

---

**THE JAPANESE CHAPTER**

Chair: Osamu Watanabe
Vice Chair: Kazuhisa Makino
Secretary: Ryuhei Uehara
Email: eatcs-jp@is.titech.ac.jp
URL: http://www.misojiro.t.u-tokyo.ac.jp/EATCS-J/index.html

---

4http://www.kurims.kyoto-u.ac.jp/~takazawa/KPSW/index.html
FSTTCS and IPEC 2010  The 30th edition of FSTTCS, the annual conference of the Indian Association for Research in Computing Science (IARCS) was held at the Institute of Mathematical Sciences, Chennai from December 15–18, 2010. The conference was colocated with the 5th International Symposium on Parameterized and Exact Computation (IPEC 2010, formerly IWPEC), which was held before FSTTCS from December 13–15, 2010.

The Programme Committee for FSTTCS 2010 was chaired by Kamal Lodaya and Meena Mahajan from the Institute of Mathematical Sciences, Chennai. The invited speakers for FSTTCS 2010 were Rajeev Alur (U Penn, USA), Bruno Courcelle (LaBRI, Bordeaux, France), Pavel Pudlák (Math. Institute, Academy of Sciences, Czech Republic), Santosh Vempala (Georgia Tech, USA) and Wieslaw Zielonka (LIAFA, Paris 7, France).

The Programme Committee for IPEC 2010 was chaired by Venkatesh Raman and Saket Saurabh from the Institute of Mathematical Sciences, Chennai. The invited speakers for IPEC 2010 were Anuj Dawar (Cambridge, UK), Fedor Fomin (Bergen, Norway) and Toby Walsh (New South Wales, Australia).

FSTTCS 2011 will be held in IIT Bombay. The Programme Committee will be chaired by Supratik Chakraborty (IIT Bombay) and Amit Kumar (IIT Delhi). The call for papers will soon be available at http://www.fsttcs.org.

TECS Week 2011  The ninth TCS Excellence in Computer Science Week (TECS Week 2011) was held at TRDDC, Pune, from 3–7 January. It was jointly conducted by Tata Research Development and Design Centre (TRDDC) and the Indian Association for Research in Computing Science (IARCS).
TECS Week is an annual event featuring a series of lectures on a topic related to computer science and engineering. It aims at providing high-quality computer science education to students, faculty and practitioners from developing countries. The topic for this year’s TECS Week was *Computer Networks*. This year there were 90 participants.

The speakers at TECS Week 2011 included Albert Greenberg (Microsoft, USA), Shivkumar Kalyanaraman (IBM Research, Bangalore, India), Vitaly Shmatikov (University of Texas, Austin, USA), and Dr. K K Ramakrishnan (AT&T Labs Research, USA). In addition, this year’s program also included guest lectures by V.S. Subrahmanian (University of Maryland, College Park, USA), Natarajan Shankar (SRI International, USA) and Pravin Bhagwat (CTO, AirTight Networks).

**Mysore Park Workshop on Algorithms and Complexity** The workshop on *Recent advances in Algorithms and Complexity* was held during October 21-24, 2010 at the Mysore Park campus of Infosys. There were around 60 participants from various academic institutes and research labs in India. The aim of this workshop was to bring together researchers and students working in algorithms and complexity in India in order to discuss recent developments in this area. The organisers were Manindra Agrawal (IIT Kanpur) and Kavita Tellekapalli (TIFR, Mumbai).

The main talks held here were the following:

- **A recent attempt at proving $P \neq NP$** by Manindra Agrawal (IIT Kanpur).
  Manindra discussed some details of the flawed proof of $P \neq NP$ claimed by Vinay Deolalikar, and also summarized the events in the weeks following the announcement of the “result”.

- **A new breakthrough result on Steiner tree approximation** by Naveen Garg (IIT Delhi).
  Naveen presented an improved LP-based approximation for Steiner trees by Byrka, Grandoni, Rothvoss, and Sanita which won a best paper award at *STOC 2010*.

- **Recent advances in communication complexity** by Navin Goyal (IBM IRL, Delhi) and T.S. Jayram (IBM Almaden, USA).
  The talks by Navin and Jayram looked at two recent results in the field of communication complexity. Both problems discussed were in the framework of the two-party model, where two parties have to compute a function $f(x, y)$, but each only holds a part of the input. The common idea in the two
results was to use information theoretic techniques to obtain bounds on the amount of communication required.

- A new faster approximation for max flow by Nisheeth Vishnoi (MSR, Bangalore).

Nishit spoke on the recent paper “Electrical Flows, Laplacian Systems, and Faster Approximation of Maximum Flow in Undirected Graphs” by Christiano, Kelner, Madry, Spielman and Teng on faster approximation algorithms for max flow in undirected graphs. The main ingredients in the new algorithm are the use of electrical flows and the near linear time algorithm of Spielman and Teng for solving symmetric, diagonally-dominant linear systems approximately.

- A survey talk on subexponential parameterized algorithms by Venkatesh Raman (IMSc, Chennai).

Venkatesh gave a survey talk on subexponential parameterized algorithms. Some major open problems in this field are subexponential algorithms for directed \(k\)-paths, Steiner trees on planar graphs and better algorithms for feedback vertex set in general directed graphs.

- Matrix norms and communication complexity by Satya Lokam (MSR, Bangalore).

Sayta spoke on matrix norms and communication complexity. He reviewed the analytical methods of Sherstov, Linial, Shraibman to show the strict containment of the following classes: \(P^{cc} \subset BPP^{cc} \subset PP^{cc} \subset UPP^{cc}\).

Mysore Park Workshop on Concurrent and Distributed Computing The next Mysore Park Workshop, entitled The Chemistry of Concurrent and Distributed Programming, will be held in mid-February, 2011. The workshop will focus on issues related to the development of distributed and concurrent systems. Topics of interest include correctness criteria, specification mechanisms, design patterns, platform independence, verification and synthesis. The workshop is organized by Madhavan Mukund (CMI, Chennai), Madan Musuvathi (MSR, Redmond, USA) and Ganesh Ramalingam (MSR, Bangalore). More details about the workshop can be found at http://research.microsoft.com/en-us/um/redmond/events/mysorepark/ccdp11.htm.
Acknowledgment  The report on TECS Week 2011 was provided by Shirish Karande and Rekha Tulsani, TRDDC, Pune. Kavita Tellekapalli (TIIFR, Mumbai) contributed the material on the Mysore Park Workshop on Algorithms and Complexity.

Madhavan Mukund, Chennai Mathematical Institute
Secretary, IARCS (Indian Association for Research in Computing Science)
http://www.cmi.ac.in/~madhavan
1 Scientific and Community News

0. The Twelfth Asian Logic Conference will be held in Wellington, New Zealand from 15–20 December 2011. This meeting will be held jointly with a meeting of the Australasian Association for Logic (AAL), http://www.victoria.ac.nz/victoria-conferences/conference.aspx?p=54&n=Asian%20Logic%202011.

1. The latest CDMTCS research reports are (http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/secondcgi.pl):

392. A.A. Abbott and C.S. Calude. Von Neumann Normalisation of a Quantum Random Number Generator. 10/2010


2 A Dialogue with Professor Joseph F. Traub

Joseph F. Traub is the Edwin Howard Armstrong Professor of Computer Science at Columbia University and External Professor, Santa Fe Institute http://cs.columbia.edu/~traub. He is the author or editor of ten monographs and some 120 papers in computer science, mathematics, physics, finance, and economics. In 1959 he began his work on optimal iteration theory culminating in the 1964 monograph which is still in print. Subsequently he pioneered work with Henryk Wozniakowski on optimal algorithms and computational complexity applied to continuous scientific problems (information-based complexity). He collaborated in creating significant new algorithms including the Jenkins-Traub algorithm for polynomial zeros, as well as the Kung-Traub, Shaw-Traub, and Brent-Traub algorithms. One of his current research areas is quantum computing. From 1971 to 1979 he headed the computer science department at Carnegie Mellon University and led it from a critical period to eminence (see Joseph Traub digital archive at CMU http://diva.library.cmu.edu/traub). From 1979 to 1989, he was the founding chair of the computer science department at Columbia University. From 1986 to 1992 he served as founding chair of the Computer Science and Telecommunications Board, National Academies, and served as chair again 2005 – 2009. Traub was founding editor-in-chief, Journal of Complexity, in 1985, and continues in that capacity.

His numerous honors include election to the National Academy of Engineering in 1985, the 1991 Emanuel R. Piore Gold Medal from IEEE, and the 1992 Distinguished Service Award from the Computer Research Association (CRA). He is a Fellow of the Association for Computing Machinery (ACM), the American Association for the Advancement of Science (AAAS), the Society for Industrial and Applied Mathematics (SIAM), and the New York Academy of Sciences (NYAS). He has been Sherman Fairchild Distinguished Scholar at the California Institute of Technology, and received a Senior Scientist Award from the Alexander von Humboldt Foundation. He was selected by the Accademia Nazionale dei Lincei in Rome to present the 1993 Lezione Lincee, a cycle of six lectures. Traub received the 1999 Mayor’s Award for Excellence in Science and Technology. The award was presented by Mayor Rudy Giuliani at a ceremony in New York City. In 2001 he received an honorary doctorate of science from the University of Central Florida.
Erol Gelenbe: To what do you ascribe your very successful career?

Joseph Traub: The short answer is mostly just plain dumb luck. Of course I also took advantage of some opportunities. I’ll give you some examples. I entered Columbia in 1954, intending to take a Ph.D. in theoretical physics. In 1955, on the advice of a fellow student, I visited IBM’s Watson Laboratories at Columbia. At the time this was one of the few places in the country where a student could get his hands on the computer. I was hooked. Due to that piece of luck, I’ve spent the last 55 years involved with computation. My Ph.D. thesis on computational quantum mechanics was done on the IBM 650, a 2000-word drum memory machine. I believe that the need to be very economical on this computer may have led to my early interest in optimal algorithms and computational complexity.

The next stroke of luck was being hired in the research division of Bell Laboratories in 1959. This was a golden age at the Labs. You were free to work on whatever interested you; if your work had impact on the company, all the better. I had the freedom to spend some four years doing research which culminated in the creation of optimal iteration theory and the publication of a monograph in 1964. If I had been an assistant professor at a university, it would have been very dangerous to create a new area while publishing very little, but I could do it at Bell Labs.

The next stroke of luck occurred during a sabbatical at Stanford in 1966. I met a student, Michael Jenkins, who was looking for a Ph.D. advisor. We developed the Jenkins-Traub algorithm as well as high-quality portable software for polynomial zeros.

In 1970 I was at the University of Washington. I advertised for a GRA, and interviewed about a dozen students. I selected H. T. Kung, and the following year, brought him with me to CMU. He eventually joined the CMU faculty and today is a chaired professor at Harvard.

In the spring of 1971, I was selected to be head of the computer science department at CMU. I was 38 years old and had this opportunity because I’d gotten into the field so early. If I’d been in a mature field, I would never have had such an opportunity. Alan Perlis, the department head, was leaving to become founding chair of the department at Yale. Al didn’t publish much but was a towering figure at CMU. Allen Newell, Alan Perlis, and Herbert Simon had founded the department in 1965. Perlis and I overlapped by just a few days but he gave me invaluable advice. One of the things I recall is that he advised me to rapidly tenure Bill Wulf, who was then an assistant professor. Soon Bill was a full professor, and he later succeeded me as chair of the Computer Science and Telecommunications Board of the National Research Council, and then became President of the National Academy of Engineering.

The department was quite small, including Gordon Bell, Nico Habermann,
Allen Newell, Raj Reddy, Herbert Simon, and William Wulf. Just prior to 1971, many faculty had left the department to take positions elsewhere. Those professors who remained formed a core of world-class scientists recognized as leaders of the discipline. I worked with the faculty to recruit new members and diversify research funding. I was deeply concerned whether we would remain a leading department. Perhaps it’s just as well that I didn’t know of a commitment made by the senior faculty to stay at CMU for at least one year to see if the department could be turned around. I often had the feeling that the department and I had been created for each other. By the time I left Carnegie in 1979, we had some fifty teaching and research faculty.

Another opportunity occurred in 1972. Lawrence Livermore Laboratory planned to acquire a STAR computer and hired me as a consultant. I became fascinated with parallel computing, which I saw as a very interesting new direction for computing. Perhaps I was too early. I remember giving a talk at a major research university on why parallel computing was going to be very important. The first question after the lecture was from a very well known professor: Joe, you don’t really think we’re ever going to use these computers to solve our problems?

In 1972 occurred a stroke of luck that was to change my scientific work. I received a registered package containing a paper and letter from someone named Henryk Wozniakowski in Warsaw. A Polish professor had pointed him to my 1964 monograph. The paper proved conjectures I’d framed in the 1964 book but in much greater generality. Henryk visited me at CMU in 1973 and that was to be the beginning of a collaboration that has extended for almost forty years. We were to start and build the field of information-based complexity.

By 1979 the department was thriving by every measure. I had been head at CMU for seven years, and I could think of moving on to new challenges. Then came the next stroke of good timing. With some exceptions, the Ivy League universities had lagged in the building of computer science departments. Now Columbia decided to start a department, and invited me to return and to build it. I accepted, and went to Columbia in 1979.

By the mid-80s many papers were being written on information-based complexity and there was no obvious place to publish them. I was not particularly interested in starting a journal, but there was a need. I noted with surprise that there was, as far as I could tell, no journal with the word complexity in the title. In 1985, I started the Journal of Complexity. It’s now in its 26th year and is much broader than only publishing papers in information-based complexity.

In 1986, a different opportunity came knocking. I was asked to start a computer science board at the National Academy of Sciences. More precisely, it was a board of the National Research Council, which is the working arm of the National Academies. I was told two previous boards had failed—I was determined there would not be a third failure. I called the new board The Computer Science
and Technology Board (CSTB) and appointed leaders from academia and industry to serve as members. About a year after the establishment of CSTB, I was fortunate to hire a superb staff director, Marjory Blumenthal. Around 1988, Frank Press, the president of the National Academy of Sciences, told me that the board on telecommunications (BOTCAP) was failing, and that the decision had been made to terminate it. He asked us to add telecommunications to our responsibilities. Marjory and I wanted to preserve the abbreviation CSTB, so we renamed the board the Computer Science and Telecommunications Board, which has remained its name to the present.

Another opportunity came in 1990, when I was asked to teach in the summer school of the Santa Fe Institute (SFI). I was greatly stimulated by the variety of topics studied at SFI and have been associated with the Institute ever since. Currently I'm serving as an external professor.

In the early 90s, we were lucky to be given a collateralized mortgage obligation (CMO) by Goldman Sachs. This involved computing integrals in 360 dimensions. A Ph.D. student, Spassimir Paskov, computed the integrals by Monte Carlo (MC) and quasi-Monte Carlo (QMC). To our surprise, and later to the surprise of Wall Street, QMC always beat MC by one to three orders of magnitude.

Of course the greatest stroke of luck of all was meeting and marrying Pamela McCorduck. I’m also blessed with two loving children, Claudia and Hillary, and four wonderful grandchildren.

EG: You’ve done a great deal of research. Could you tell us about some of your early work?

JT: Shortly after I joined Bell Labs in 1959, a colleague asked me how to compute the numerical solution of a certain problem, which involved the solution of a complicated nonlinear equation. I could think of a number of ways to solve the problem. What was the optimal algorithm, that is a method that would minimize the required computational resources? To my surprise, there was no theory of optimal algorithms. (The phrase computational complexity which is the study of the minimal resources required to solve computational problems was not introduced until 1965.) I set out to construct a theory of optimal algorithms for the solution of the nonlinear equation \( f(x) = 0 \). I had the key insight that the maximal order of an iteration depended on the available information about \( f \) and not on the structure of the iteration. (Maximal order is closely related to computational complexity.) This was such a powerful idea that I was sure someone else would announce it. I scanned the world literature, fearing such a publication. To my immense relief, no one published this idea. Wouldn’t it be useful for researchers to be automatically notified of papers in which they would be interested? This led to work on such a publication notification system [1, 3].

I first presented the work on what was to become optimal iteration theory
at the 1961 National ACM Conference [17]. In those days the national ACM conference was a big deal, and many of the researchers in computing attended. I kept building the theory and soon had a manuscript of some 120 pages. Being a very young and naive researcher, I sent this manuscript to Mario Juncosa, the editor-in-chief of the Journal of the ACM, who wrote back that it would be a long time before he read the manuscript. Just then, Prentice-Hall asked me if I wanted to write a book. I set to work and wrote the monograph, Iterative Methods for the Solution of Equations [18]. The editor of the Prentice-Hall Series in Automatic Computation was George Forsythe, who was to become founding chair of the Computer Science Department at Stanford. I’m pleased that this book is still in print, going on fifty years since its publication. The book would have been better titled Optimal Iteration Theory. It marked the beginning of lifetime work on optimal algorithms and computational complexity for continuous problems. The Introduction begins with: The general area into which this book falls may be labeled algorithmics. By algorithmics we mean the study of algorithms... Don Knuth credits me with coining algorithmics [6].

So far I’d worked on general nonlinear equations. For such problems, convergence could not be guaranteed. Was there a class of nonlinear equations for which one could guarantee convergence? The answer is yes, for polynomial equations [19]. In 1966 I was a visiting professor at Stanford, where I met a Ph.D. student, Michael Jenkins. We continued the work on global convergence which led to what is usually called the Jenkins-Traub algorithm [5]. The algorithm consists of three stages, of which the third is the most important. It can be shown that this stage is equivalent to applying Newton iteration to a sequence of rational functions, which is converging to a first-degree polynomial whose zero is one of the desired answers. Although Newton iteration requires the evaluation of a derivative at each step, the Jenkins-Traub algorithm does not require the evaluation of any derivatives. It can be shown that under mild conditions, the algorithm always converges, and that the rate of convergence is faster than the quadratic rate of Newton. See the Wikipedia article [34] for more. Jenkins wrote a high-quality portable program implementing the algorithm. This algorithm is still one of the most widely used methods for this problem and is included in many textbooks.

In 1972 I became fascinated with parallel computing and organized a symposium at CMU in 1973, which may have been the first on this subject [20]. I spoke on this topic on the 1974 IFIP Congress [21].

In the 70s I became interested in algebraic complexity. Mary Shaw was a student in my class when I spoke about Horner’s method for evaluating polynomials, which was known to be optimal. I conjectured that if one wanted to evaluate all the derivatives of a polynomial, the optimal method would take a quadratic number of multiplications. Mary showed me that she could beat that, and we worked together to get the number of multiplications to linear [16]. Mary is now the Alan...
Next, my student H. T. Kung and I showed that computing the first \( n \) terms of any algebraic function was no harder than multiplying \( n \)th degree polynomials [7]. This problem has a long history; Isaac Newton missed a key point.

Richard Brent and I were able to show that computing the \( q \)-th composite of a power series was no harder than computing a single composition where \( q \) is any number [4]. One day, I got a call from Don Knuth who had heard about our work. I told him I could mail him a preprint in a few days. Don replied that he didn’t want to wait, since he was working on that part of his book just then. He had redone parts of our analysis and just wanted to check that it agreed with what we had done.

This ended my work on algebraic complexity because I was about to move into an entirely new direction.

**EG:** Can you tell me what happened next?

**JT:** I mentioned earlier that Henryk Wozniakowski visited me at CMU in 1973. That was to be the beginning of a collaboration that has spanned almost forty years. Initially, we continued the work on optimal iteration theory. Then in 1976 there came an event that changed the course of our research. A Ph.D. student named Arthur Werschulz, now a professor at Fordham University and part of our research group at Columbia University, gave a seminar, where he used some of the techniques from nonlinear equations to attack the complexity of integration. Our reaction was that integration is inherently different from solving nonlinear equations; one doesn’t solve integration iteratively. Because these problems are so different, there must be a general structure that underlies this and many other problems. Our search for the general structure led to our monograph [29]. We called this new field analytic complexity. This was to differentiate it from algebraic complexity, which was a very active research area in the late 60s and 70s.

Algebraic complexity deals with algebraic problems such as the complexity of matrix multiplication, where information about the input is complete, while analytic complexity deals with problems from analysis, such as the complexity of high-dimensional integration, where information about the continuous input is partial. Let me elaborate this last point. In calculus, students are taught to compute univariate integrals exactly. But most integrals cannot be expressed in terms of elementary functions; they have to be approximated numerically. This is especially true of real world high-dimensional integrals, such as the integrals common in mathematical finance. We sample the integrand; that is why the information about the mathematical input is partial. Other problems studied in analytic complexity include optimal algorithms and computational complexity of systems of ordinary differential equations, high-dimensional approximation, partial differential equations, continuous optimization, and nonlinear equations.
Greg Wasilkowski joined Henryk and me to write the monograph [30]. We renamed the field epsilon-complexity. One day, Pamela asked me why epsilon-complexity? I replied that epsilon denotes a small quantity and that it measures the error in the answer. She did not seem impressed. Since Pamela is the author of numerous books, I took her lack of enthusiasm seriously and started thinking about a new name. One day I was chatting with my friend, Richard Karp, who, as you know, was a pioneer in the study of NP-completeness. He suggested information-based complexity, which we adopted as the name of the field. It was the name of our monograph [31]. For brevity we often refer to the field as IBC. Typically, IBC theory is developed over abstract linear spaces such as Hilbert or Banach spaces. The applications are often for problems with a very large number of variables.

Because the information is partial, IBC is able to use powerful adversary arguments at the information level. The general idea behind an adversary argument is the following: the adversary creates a situation where the inputs are indistinguishable but the outputs are quite different. It is therefore impossible to compute a good approximation because if we claim an approximation to one output as the answer, the adversary will say the second output is the correct one. Adversary arguments are often used to find a good lower bound on the information complexity and hence tight lower bound on the computational complexity (see, for example, [15], section 2). This may be contrasted with the rest of theoretical computer science where researchers work on discrete problems with complete information and have to settle for conjectures on the complexity hierarchy. We find these adversary arguments very natural but we’ve learned that this way of thinking is so different that many of our colleagues in theoretical computer science find them difficult. See [28] for an expository account of IBC.

IBC has grown vastly over the past twenty years. See [26] for a brief history and [9] for a survey.

In the early 1990s, I had a Ph.D. student named Spassimir Paskov. Spassimir was very strong in theory, but I wanted to broaden him. We had gotten a collateralized mortgage obligation (CMO) from Goldman Sachs. (A CMO is a bond that represents claims to specific cash flows from large pools of home mortgages.) This involved computing integrals in 360 dimensions. I asked Spassimir to compute the integrals using quasi-Monte Carlo (QMC) and Monte Carlo (MC). It was believed by experts that QMC, which uses deterministic sampling, was not good for dimension greater than 12. To the amazement of our research group, Paskov reported that QMC beat MC by one to three orders of magnitude. The results were presented to a number of Wall Street firms, who were initially skeptical. Other researchers then got similar results. QMC is not a panacea for all high dimensional integration. It is still an open question to explain why QMC is superior to MC for financial instruments. See the Wikipedia article Quasi-Monte Carlo Methods in
Finance for a survey [35].

Moore’s Law, which has explained the exponential increase in computer power over some five decades, is coming to an end. Starting in 2001 our research group has been applying IBC ideas to solve continuous problems such as the Schrödinger equation and path integrals on quantum computers. Among other objectives we want to answer the question posed by Nielsen and Chuang [8]. Of particular interest is a decisive answer whether quantum computers are more powerful than classical computers. To answer this question, one must know the classical and quantum complexities, which can sometimes be obtained using IBC techniques. A survey on solving continuous problems on a quantum computer may be found in [14].

EG: Can you tell me about some of the organizations you’ve built?

JT: As I mentioned earlier, by getting involved in computing so early I had opportunities I would not have had in a mature discipline. The first was being selected as head of the computer science department at CMU when I was 38 years old. As I told you earlier, the department was very small but the faculty formed a core of world-class scientists. Crucial was adding outstanding faculty and diversifying research funding. We decided to revamp the Ph.D. program. One of the innovations was the creation of the Black Friday Meeting, which was held at the end of each semester. The entire faculty reviewed every Ph.D. student. Every student received a letter regarding his or her progress. I thought this was a very effective management tool. Allen Newell, Herb Simon and I talked about the greening of CMU and Pittsburgh, using computers. That has come to pass, big time.

In 1979, Columbia invited me to start a new computer science department. At the time there were two efforts in computer science: an Electrical Engineering and Computer Science Department and a group in the Statistics Department. The two groups were at loggerheads, and unable to recruit good junior faculty. The plan was that both these efforts were to be terminated.

I was able to negotiate some very good things for the new department. We would get our own building, and I would select the architect. I pointed out that the teaching load in computer science departments at leading private research universities was one course a semester, and got the same at Columbia. I accepted the position and started on July 1, 1979.

It was to be the toughest challenge I ever had. At CMU, the department had enough DEC computers to heat the building. At Columbia, the entire engineering school had a single DEC machine; an 11/45 model, as I recall. There were three tenured faculty inherited from the terminated efforts, as well as a number of junior faculty. None of the junior faculty belonged in a department with national ambitions; they were all gone within two years. I set out to hire outstanding new Ph.D.s. Since many universities and corporate research laboratories were
hiring at this time, the competition was very tough, but we succeeded in hiring some outstanding young faculty. With almost no faculty, we were trying to teach several thousand students who were taking our courses. I didn’t advertise to our newly-hired hotshots that they’d be teaching some 200 students per course.

But we had some great successes. We received a substantial grant from IBM. I took the new faculty to meet Bob Kahn, the DARPA IPTO director. He was so impressed with the new faculty that he decided to give us major funding. Furthermore, for the first time Columbia had a connection to the ARPAnet. We started bachelors, masters, and Ph.D. programs, and taught computer science to all of Columbia University.

As I mentioned earlier, I started the Journal of Complexity in 1985. In the early years, all papers funneled through me. I realized I was a bottleneck, so I created an Editorial Board, who could independently accept or reject papers or require revisions. The journal is going strong in its 26th year.

In 1986 I was asked to create a computer science board for the National Research Council (NRC). It’s now called the Computer Science and Telecommunications Board (CSTB). Our first report, for which the late Michael Dertouzos did much of the writing, was called The National Challenge in Computer Science and Technology. We had a fairly difficult time getting it through the very thorough NRC review process. Much of this report was devoted to policy, whereas my impression was that NRC was more comfortable with technology. CSTB continued to work on policy as well as technology, and in time, that became highly appreciated at the NRC. I rotated off CSTB in 1992 and then served as chair again in 2005 – 2009. To see what reports CSTB has completed and what projects are currently underway, visit www.cstb.org.

I had a hand in building four organizations: the CS department at CMU, the CS department at Columbia, the Journal of Complexity, and the Computer Science and Telecommunications Board. The common ingredient for success was excellent people.

EG: You mentioned your role in building organizations. Did you also play a role in the creation of other entities?

JT: Because I got into computing so early, I had such opportunities. For example, I was one of the founders of the Computer Research Association (CRA) in 1972. We decided to create what became the Federated Computing Research Conference (FCRC) at a meeting which I believe was held in Washington, D.C. I was a founding member of the scientific advisory committee (ISAT) of DARPA in 1986.

EG: Do you have any regrets about something you did not pursue?

JT: There is something important I should have done. Starting in 1985, I noticed various ways in which our information infrastructure was vulnerable to electronic
or physical attack. I imagined myself to be a terrorist, or an enemy country, and targeted aspects of what we would today call the national information infrastructure. I felt it was just because we were the most advanced country in our use of information technologies, we were and are the most vulnerable. I also felt we were vulnerable to physical attack. Let me give you an example. I was given the opportunity to visit the floor of the New York Stock Exchange one morning just before it opened. The only visible security was one guard, equipped with a revolver. There may, of course, have been security that was not visible to me, but I doubt it. I thought what a tempting target the symbolic heart of our capitalist society this would make, and the damage a couple of hand grenades would inflict. The actual processing of trades was executed across the river in Brooklyn but I doubted that it was sufficiently secure against physical or electronic attack.

I did not go public with my concerns because I was worried about giving individuals or countries ideas. That was foolish; our enemies are very smart. I now feel I should've spent a considerable amount of my energy and time alerting the country.

I finally went public when I gave the keynote address at a symposium at the National Academy of Sciences celebrating the tenth anniversary of CSTB in 1996 [23]. I pointed out the vulnerability of what I called the virtual estate, which consists of bank accounts, equities, CDs, pension accounts, etc. I called it the virtual estate because it’s recorded in electrons. If you were a terrorist, and wanted to do a great deal of damage to American institutions and individuals, a natural target would be the virtual estate. Our virtual estate is just one example of a potential target. Others include the power grid, and our communications systems.

Who should be in charge of protecting our infrastructure? I argued for strong Federal government leadership, centered in the executive branch.

EG: Can you say something about the future of computing, especially as it relates to your interests?

JT: That is such a deliciously open-ended question–I’ll confine myself to just four issues.

The first has to do with scaling laws. Perhaps the most famous scaling law is Moore’s Law, an empirical law which has driven computing for almost half a century. Moore’s Law is running up against a number of fundamental physical limits. For a while we will benefit from multicores, many cores, and massive parallelism. But there are considerable impediments to parallel computing. Parallel machines are difficult to program, and some problems are difficult to decompose. That is why there’s much interest in radically different kinds of computing, such as quantum, photonic, molecular, and biological computing. Of course Moore’s Law is not the only important scaling law. Another example is the doubling rate of bandwidth, which is much shorter than that of chip density. How should we
plan our computing and networking in light of this effect?

A second area is information-based complexity (IBC). Quite a few years ago, I ended an MIT lecture by stating some open questions. Afterwards, Marvin Minsky told me he saves his good questions for his students. I replied that there were lots more where those came from. That’s always been the case in IBC. We end many papers and talks with a list of open problems. For example, Erich Novak and Henryk Wozniakowski are writing a three-volume monograph, Tractability of Multivariate Problems, Volume I [10] listed thirty open problems, while Volume II, which has just been published, has sixty-one more. Volume III, which is expected in 2012, will list many additional open problems. Why are there so many open problems? I believe it’s because we’re asking new questions about many continuous scientific problems, a vast domain. Furthermore, when the technology changes, or might change, that alters what algorithms are permitted. A good example is quantum computing, where, for example, we’re investigating the power of quantum computing for solving the problems of quantum mechanics.

A third area is cybersecurity, far more than just the protection of the virtual estate. Two very different issues have recently been studied by the Computer Science and Telecommunications Board. One is deterrence strategies for the U.S. government towards preventing cyberattacks. A recent letter report on this topic is now available; see www.cstb.org. Another is in regard to U.S. acquisition and use of cyberattack capabilities; a recent CSTB report is also available. Cybersecurity can only become still more important.

I’ll end with concern about computer science majors. There is an odd dissonance between the feelings of prospective students and their parents, on the one hand, and university computer science faculty, on the other. Students seem reluctant to study computer science for two reasons: concern jobs will be outsourced; and a feeling that the big advances are behind us. But my colleagues and I feel that computer science is more exciting than ever. Bill Gates has expressed concern about not being able to hire American computer science majors. We worry that women are not attracted to computer science, which cuts us off from half the brains of the country. This is at the very time that China is making huge investments in computer science education. The key to the country’s future is innovation, and it’s vital that computing attract some of our country’s best and brightest.

Cristian Calude: In your 1998 paper Non-Computability and Intractability: Does It Matter to Physics? you write I’m not convinced that non-computability need be of concern. What is your current position regarding this statement.

JT: I will briefly summarize the issue for the benefit of the reader. See also [28], Chapter 9. Consider, for example, partial differential equations with computable initial conditions but non-computable solutions. The equations can be very sim-
Examples are the wave equation with initial conditions which are not twice differentiable and the backwards heat equation. The renowned physicist and mathematician Roger Penrose is concerned by the result that the wave equation with computable initial conditions can have non-computable solutions; he called this a rather startling result, [11]. Is this really a startling result? Let’s have a deeper look.

The differential equations mentioned about are special cases of ill-posed equations. Werschulz [33] proved that if a problem is ill-posed it is impossible to compute an \( \varepsilon \)-approximation to the solution at finite cost even for arbitrarily large error \( \varepsilon \). But this is a worst-case result. There is a surprising result that every ill-posed problem is well-posed on the average for every Gaussian measure; see [27] for a survey of the work leading to this result. Thus the non-solvability of ill-posed problems is a worst-case phenomenon. It melts away in the average case for reasonable measures. Nothing has happened to make me change my mind that non-computability may not be a cause for concern for physicists. As in the above example it may simply be a worst case phenomenon.

CC: It seems to me your concern was that non-computability could be bad for physics. What about the possibility of being an asset?

JT: I agree that non-computability can cut in two ways. Its analogous to the situation in cryptography. We want secure encryption to protect out private information. On the other hand, we want to read encrypted messages between terrorists to foil the planning of attacks.

CC: Starting with Ralph Gomary’s tripartite division of science into the known, the unknown which may someday became known, and the (most interesting) unknowable, the part which will never be known you’ve written My goal is to move the distinction between the unknown and the unknowable from philosophy to science. Did you make further steps towards achieving this goal, and if yes, can you summarize them?

JT: I’d like to begin with some background regarding my interest in this issue. That quote comes from [25]. The first time I wrote about this issue was in 1991 [22]. I gave a talk on What is Scientifically Knowable at a symposium celebrating the 25th anniversary of the Computer Science Department at CMU and this reference is in the anniversary commemorative. Gödel’s work has had a profound impact on mathematics. It established fundamental limits on mathematical proofs. It was an enrichment of mathematics. I hoped that establishing limits to science by proving that the answers to certain scientific questions were unknowable would be an enrichment of science. How might one prove that the answer to a question is unknowable? I’ve proposed several possible attacks. The first attack is the follow-
ing. A scientific question does not come equipped with a mathematical model.
Researchers develop models for scientific questions. Consider then all formal models that capture the essence of a scientific question. Prove that every formal model is undecidable, or computationally intractable. It seems to me that would be one way of proving the answer is unknowable. However, although this might be a possible attack in principle it is far from evident that it could actually be carried out for any nontrivial question. A different attack is proposed in [24]. Rather than a direct attack, which considers all mathematical models, perhaps an indirect attack would have more chance of success. Computational complexity might serve as a guidepost. The intractability theorems of information-based complexity are not proven by varying algorithms. Instead general theorems are proven from which we can infer intractability of specific mathematical problems. Can this procedure be adapted to derive negative properties (undecidability, intractability) which any mathematical model, for a certain scientific question, must possess? I still think the question of how to distinguish the unknown from the unknowable is an interesting question. However, since the papers I wrote in the 90’s I’ve moved into other research areas.

References

The Bulletin of the EATCS


[34] Jenkins-Traub algorithm, Wikipedia.
The EATCS Columns
This is my last issue of the complexity column. After editing it for seven years I think it is time that someone else brings new energies to the column and V. Arvind from the IMSC Chennai (arvind@imsc.res.in) will be the new editor. I wish him the best and would like to thank again all the authors that have contributed to the column in the time I have been editing it. These were: Jorg Flum, Martin Grohe, Leen Torenvliet, Harry Buhrmann, Peter, Hoyer, Robert Spalek, Pascal Tesson, Denis Thérien, V. Arvind, Johannes Köbler, Wolfgang Lindner, Venkatesan Guruswami, Meena Mahajan, Gorjan Alagic, Alexander Russel, Nikolai Vereshchagin, Alexander A. Sherstov, Fabian Wagner, Beate Bollig, Nitin Saxena, Arkadev Chattopadhyay, Kazuyuki Amano, Michael Thomas, Heribert Vollmer and Claus Diem. They have written very high quality articles, making it possible for a general audience to have a look into different regions of the complexity theory landscape and I am very grateful to them. In my last issue we come back to the basics. Claus Diem provides a close look to the central notion of bit complexity.
ON THE NOTION OF BIT COMPLEXITY

Claus Diem *

Abstract

In many works in the fields of computational complexity, algorithmic number theory and mathematical cryptology as well as in related areas, claims on the running times of algorithms are made. However, often no computational model is given and the analysis is performed in a more or less ad hoc way, counting in an intuitive way “bit operations”. On the other hand, the computational model of a successor RAM with logarithmic cost function provides an adequate and formal basis for the analysis of the complexity of algorithms from a “bit oriented” point of view.

This motivates the search for a result on the simulation of machines in a suitably defined general model by successor RAMs. In this work, a very general RAM model is defined, and then a “quasi-optimal” result on the simulation of such machines by successor RAMs is given.

1 Introduction

In a large body of works in the fields of computational complexity, algorithmic number theory and mathematical cryptography as well as in related areas, claims on the running times or time complexity of algorithms are made. However, in a substantial part of these works, the analysis of the algorithms is performed in a more or less intuitive and ad hoc way without reference to a specific model of computation.

Often, the running time (or expected running time) is computed by counting in some intuitive way “bit operations”. Or to phrase it differently: in a certain intuitive way, the bit complexity of algorithms is considered. Such an approach is clearly sufficient if one is merely interested in questions on complexity from a “qualitative” point of view (disregarding exponents) – as is often the case in complexity theory. However, often more concrete statements are made, and then the question poses itself whether the claimed running time holds true in a particular “bit-oriented” model of computation.

*Universität Leipzig, Mathematisches Institut, diem@math.uni-leipzig.de
The Bulletin of the EATCS

The situation concerning space requirements or space complexity is similar but in fact – as we will discuss below – it is even worse because it is even less clear what is exactly meant by claims concerning space requirements.

This is by far a new phenomenon. Already in 1980 Arnold Schönhage observed; see [5]:

Many of the concrete algorithms given in the literature are (at least implicitly) designed for multitape Turing machines, sometimes the higher flexibility of random access machines (with a variety of instruction sets) is required, and frequently it is totally left to the reader’s imagination what the model of computation should look like.

Let us fix the following terminology: By a machine type we mean a type of Turing machines, random access machines, etc. A model of computation is then a machine type together with a time and a space measure. In some cases, these measures are obvious (e.g. for Turing machines), but in other cases – in particular for RAM models – they are not, and care has to be taken which measure is used.

It is intuitively obvious that if one speaks about running time without further comments, one should have a sequential machine type with a bit-oriented storage and atomistic instruction set in mind, and the time measure must reflect the number of bit operations required. Of course, we cannot give a rigorous definition of these intuitive notions but some requirements seem to be obvious: First, the machine type must have a most reduced set of instructions. Second, the time needed for one instruction must by definition reflect the lengths of the numbers which have to be considered for its execution.

One such model is the multi-tape Turing machine model (with various similar definitions). Another model is the successor RAM with logarithmic cost measure (again with various similar definitions).

We note here that the bit-oriented point of view of this work is slightly different from the atomistic point of view in [5]. An atomistic model according to Schönhage is for example the Storage Modification Machine (SSM). However, from our point of view, the SSM model is not bit-oriented because it misses a bit-oriented (a priori) storage. Also, the successor RAM types (as defined in [5]) with uniform time measure deserve to be called atomistic but not bit-oriented, because in one

---

1 We do not give a rigorous general definition of “machine type”, and – consistent with this – we do not claim any general mathematical propositions on machine types. The mathematical propositions are rather for specific machine types.

2 We assume here implicitly that the machines have a program. Let us note here that in an obvious way, Turing machines can also be based on programs. This point of view is emphasized by Schönhage, cf. [5] and [6].
time unit arbitrarily many bits might be changed. Note here that it is shown in [5] that the SSM type (with obvious time measure) is real time equivalent to certain successor RAM types with uniform time measure.\textsuperscript{3}

In many works on the complexity of computational problems arising in algorithmic number theory, cryptography and related areas, it seems to be assumed that the underlying model is on the one hand bit-oriented and on the other hand, storage access is more or less immediate. These requirements are met by successor RAMs with logarithmic time measure. The very limited instruction set of these models does however often not make it possible to obtain the claimed running times in a straightforward way. For example, very often the algorithms and their analyzes require the presence of instructions for addition and subtraction of registers.

This situation motivates the search for a general result which transforms a result for any kind of random access machine model with “reasonable” time and space measures to a result for successor RAMs.

In this work, we give a rigorous definition of a RAM type with a very general instruction set (whose machines we call RAM with extended instruction set). With an adequate (and intuitive) time measure, we show that machines of this type can be simulated “quasi-optimally” (optimally up to “logarithmic factors”) by successor RAMs with logarithmic cost measure. Concerning space complexity, the result is “ quasi-optimal” too and in fact relates a particularly strong space measure on successor RAMs with a weak space measure for the general RAM type. The result on time complexity shows in particular that RAMs with additional instructions for addition and subtraction and / or for AND, OR and, XOR and / or for concatenation and shifts can be simulated “quasi-optimally” by successor RAMs with respect to the logarithmic cost function.

The simulation is straightforward, and in fact, it essentially already appeared in the literature before; cf. the proof of [8, Theorem 19.28]. However, an extensive search in the literature did not reveal a result as the one given in this work even for the simulation of RAMs with instructions for addition and subtraction by successor RAMs.\textsuperscript{4} It is exactly the lack of a suitable reference which motivated the author to write this work.

\textsuperscript{3}The use of the notion “real time equivalent” in [5] is different for its use at other places in the literature, e.g., in [7] and [8]. In the spirit of [7], one might say that these models simulate each other in linear time. With the definitions of [8], the time measures of the models are linearly related.

\textsuperscript{4}In [7, Theorem 2.4] a corresponding result is stated for RAMs with instructions for AND, OR and XOR. However, the very short argument in [7] does not seem to be really to the point.
The Bulletin of the EATCS

2 Basic definitions and observations

We assume that the reader is familiar with RAM models, at least on an intuitive basis. Briefly, a successor RAM type is a RAM type with only one arithmetic instruction: the computation of the successor.\(^5\)

In [5] two such types are defined, called RAM0 and RAM1. Let us recall the particular definitions in [5] on a conceptual level and compare them to other definitions of RAM types in literature.

Let us define the set of natural numbers \(\mathbb{N}\) as the set \(\{0, 1, 2, \ldots\}\). All machines defined in [5] operate on the alphabet \(\{0, 1\}\) for input and output. Each machine has an input and output tape, which are read respectively write only. Furthermore, they have a program based on some instruction set. The instruction sets contain instructions based on the codes input, output, goto, halt. The input instruction reads a bit from the input tape and – according to the bit – jumps to one of two labels. The output instruction prints a bit.

As usual for random access machines, a machine of type RAM1 has registers, an accumulator, instructions to load a fixed natural number and to load and store data directly and indirectly, and an instruction for comparison. All registers and the accumulator can store arbitrary natural numbers (or bit strings). The registers are indexed by natural numbers, and the accumulator is by definition not a register. Different from other random access machine types, the RAM1 type only has one “arithmetic” (or operational) instruction: the computation of the successor. The type RAM0 is similar to the type RAM1. The essential differences are that the type RAM0 does not have an instruction for indirect addressing but it has an additional address register instead.

Usually, random access machine types defined in the literature have additional “arithmetic instructions”. The most cited type in the literature seems to be the one by Aho, Hopcroft, and Ullman ([1]). This type has instructions for addition, subtraction, multiplication and division with remainder. Other types have instructions for bitwise AND, OR and XOR, and still others have an instruction for concatenation; cf. [8, Section 1.2].

Let a RAM0 or RAM1 \(\Pi\) be given, and let \(x\) be an input to \(\Pi\). Then there are essentially two different definitions of running time of \(\Pi\) for \(x\): the uniform and the logarithmic running time. With the uniform running time, each instruction executed is given the time 1. In order to define the logarithmic running time, we first define the size of a natural number \(n\) as 0 if \(n = 0\) and \(\lfloor \log_2(n) \rfloor + 1\) otherwise.\(^6\)

\(^5\)In [8] a machine type called successor RAM is defined which has instructions for computation of the successor and the predecessor. We do not follow this definition.

\(^6\)This definition of the size of a number follows the definitions in [8]. In [7] the size of 0 is by
Now for the logarithmic running time, each instruction not involving registers or the accumulator is given the time 1, and the instructions involving registers or the accumulator are given as time $1 + \sum$ of sizes of the numbers in the accumulator or the registers in question involved (the accumulator for comparison and computation of the successor, the accumulator and one register for direct access and the accumulator and two registers for indirect access).

We define the uniform or logarithmic time measure for $\Pi$ as the function on the natural numbers assigning to each natural number $x$ the corresponding running time of $\Pi$ upon input of $x$.

On the other hand, when we speak of the state of the machine at a particular time, we refer to the state after a particular number of operations has been executed, that is, after a particular uniform time has passed.

> From a bit-oriented point of view, the logarithmic time measure is clearly the more adequate one. After all it really measures the bits involved in the execution of a particular instruction. Because of this, in the following we base our results for time complexity on this measure. We therefore call the logarithmic measure also the time complexity and denote it by $T$.

We have already mentioned that it is shown in [5] that the two successor RAM types RAM0 and RAM1 are real time equivalent with respect to the uniform time measure. In fact, the simulation in [5] reveals that they are also real time equivalent with the logarithmic time measure, thus it does not matter which type we choose. Let us – somewhat arbitrarily – define a successor RAM as a machine of type RAM1.

There are various measures of space complexity for RAMs defined in the literature. In this work, for successor RAMs we use three space measures $S_1, S_2, S_3$ which are again functions on the inputs and are defined as follows:

Let us fix a successor RAM $\Pi$ and some input $x$ to it. Let $R_i$ be the content of register $R_i$ at time $t$. Let $u_t(i) := \text{sgn(size}(R_i))$, that is, $u_t(i)$ indicates if register $i$ is used at time $t$. Further, let

\[ b := \sup \{ i \in \mathbb{N} \mid \text{register } R_i \text{ is used during the computation} \} \]
\[ = \sup \{ i \in \mathbb{N} \mid \exists t \in \mathbb{N} : u_t(i) = 1 \} . \]

definition 1. The logarithmic function in [1] is the same as the size in [7]; see also Section 4.
We now define:

\[ S_1(x) := \sup_{t \in \mathbb{N}} \sum_{i=0}^{\infty} (\text{size}(R_i, t) + \text{size}(i) \cdot u_t(i)) \]

\[ S_2(x) := \sum_{i \in \mathbb{R}} \sup_{t \in \mathbb{N}} (\text{size}(R_i, t) + \text{size}(i) \cdot u_t(i)) \]

\[ S_3(x) := \sum_{i=0}^{b} \sup_{t \in \mathbb{N}} (\text{size}(R_i, t) + \text{size}(i)) \]

Clearly,

\[ S_1 \leq S_2 \leq S_3. \]

Measure \( S_2 \) seems to be the most accepted measure in the literature; cf. [7], [8]. Measures as \( S_1 \) and \( S_2 \) but without the term for the size of the register number are also often used in the literature. For example, in [1] the corresponding variant of measure \( S_2 \) is used. From a bit-oriented point of view, measure \( S_2 \) is however more natural.

In contrast to the definitions in [8], the space measures are also defined for inputs for which the machine does not terminate. For inputs for which the machine terminates, the measure is always finite, for inputs for which the machine does not terminate it might be finite or infinite.

Let – as defined above – \( T \) be the time complexity of \( \Pi \). Then

\[ S_3 \in O(T). \]

Indeed, let us fix some input \( x \) upon which \( \Pi \) terminates. Wlog. we can assume that numbers \( > 0 \) are only loaded directly and indirectly (no fixed number \( n > 0 \) is loaded). Then the successors of \( 0, \ldots, b-1 \) have to be computed. The logarithmic running time for this is \( \sum_{i=0}^{b-1} (1 + \text{size}(i)) \geq \sum_{i=0}^{b} \text{size}(i) \). Furthermore, if \( R_{i,j} \) is \( \neq 0 \), then at some time \( s < t \), \( R_{i,j} \) has to be stored in register \( R_i \), and the logarithmic time needed for this is at least \( \text{size}(R_{i,j}) \).

### 3 Discussion and further definitions

We now strive for a general result which transfers propositions on a type of random access machines with a very broad arithmetic instruction set to propositions for successor RAMs. The instruction set should contain all instructions of the successor RAM type and additional instructions which we call higher arithmetic.
instructions. These higher arithmetic instructions define partial functions from \( \mathbb{N}^n \) to \( \mathbb{N} \) for some \( n \), as for example do the usual addition and subtraction instructions.\(^7\)

In order that one can obtain a transfer result as desired, clearly, the partial functions defined by the higher arithmetic instructions must be computable. A subtle question is then what time and space requirements one should charge for the execution of an instruction at a particular time. Our answer to this question is to essentially the following: We again use successor RAMs to define the higher arithmetic instructions, and we measure the time and space complexities of the operations of these successor RAMs with the measures defined above.

We now describe the machines and the time and space measures we consider in detail.

First, we generalize the definition of successor RAM (i.e. RAM1) in the following way: We do not anymore have just one input tape but several input tapes. Correspondingly, the input instructions now take the following form:

\[
\text{input } m, \lambda_0, \lambda_1
\]

Here \( m \) is a natural number \( \leq \) the number of input tapes, and as before \( \lambda_0, \lambda_1 \) are labels. The operation given by this instruction is as follows: One symbol is read from tape \( m \) and then according to the symbol being 0 or 1, the program is continued at label \( \lambda_0 \) or \( \lambda_1 \).

We call the resulting machine type multi-inputtape successor RAM type (mi-successor RAM type for short).

We now define a type of computational machines which we call RAM with extended instruction set as well as time and space measures on them.

The set of instructions of the new type has two parts. The first part consists of the instructions of the successor RAM model; we call these instructions basic instructions. The second part is given as follows: For each successor RAM \( P \), we introduce an instruction \( c_P \). We call these instructions higher arithmetic instructions. The arithmetic instruction are then the instruction for computation of the successor and the higher arithmetic instructions.

The syntactic requirements for a (program of a) RAM with extended instruction set are as for successor RAMs.\(^8\)

Let now a (program for a) RAM with extended instruction set \( \Pi \) be given. Then the operation of \( \Pi \) is as follows: The basic instructions operate as usual.

---

\(^7\)The instructions one usually considers in RAM models define functions, not only partial functions.

\(^8\)One can (formally) define RAMs and programs of RAMs in such a way that a RAM and the corresponding program are (by definition) identical.
The Bulletin of the EATCS

The operation of \( c_P \) for a successor RAM \( P \) is as follows: This instruction causes \( P \) to be executed in the following way. If \( P \) has \( n \) input tapes, \( P \) takes as input the content of registers \( 1, \ldots, n \) of \( \Pi \). The output tape of \( P \) is the accumulator of \( \Pi \). If \( P \) terminates, \( \Pi \) continues with the next instruction, as usual. If \( P \) does not terminate, \( \Pi \) does not terminate either.

We define three time measures for such a machine \( \Pi \).

- **simple uniform time** simply counts the number of instructions of \( \Pi \).
- **extended uniform time** is defined as follows: The time for each basic instruction is 1, and the time for some instruction \( c_P \) is the uniform time needed for the execution of \( P \) with the inputs currently present in the respective registers of \( \Pi \).
- **extended logarithmic time** is defined in the same manner based on logarithmic time: The time for each basic instruction is measured in logarithmic time, and the time for \( c_P \) is the logarithmic time needed for the execution of \( P \) with the inputs currently present in the respective registers of \( \Pi \).

It is extended logarithmic time which captures best the intuitive idea of a bit-oriented measure for this machine type, and therefore, similarly to above, we call this measure *time complexity* and denote it by \( T \).

Let still some RAM machine with extended instruction set \( \Pi \) be given, and let \( x \) be an input for \( \Pi \). Let \( i = 1, 2, 3 \). The \( i \)th *basic space measure* of \( \Pi \) applied to \( x \) is defined as \( S_i(x) \) above applied to \( \Pi \); let us denote this measure by \( SB_i \).

We define the 1st space measure of the execution of some arithmetic instruction \( P \) at a particular (simple uniform) time of \( \Pi \) as the first space measure applied to \( P \) and the corresponding input (present in the corresponding registers of \( \Pi \)).

Now \( S_i(x) \) is the supremum of \( SB_i(x) \) and the first space measure applied to the executions of the arithmetic instructions.

The definition of the measures \( S_2 \) and \( S_3 \) is a bit more complicated: Let \( i = 2, 3 \). Let \( P \) be a successor RAM such that \( c_P \) occurs in (the program of) \( \Pi \) (it might occur several times). Then we define \( S_{i,P}(x) \) as \( S_i(x) \) above but with respect to all states of \( P \) for all executions of \( P \) during the execution of \( \Pi \). Let \( c_{P_1}, \ldots, c_{P_k} \) be all higher arithmetic instructions occurring in the (program of) \( \Pi \). Then we define \( S_i(x) := SB_i(x) + \sum_{j=1}^{k} S_{i,P_j}(x) \).

Again we have

\[
S_1 \leq S_2 \leq S_3
\]

and it is not difficult to see that

\[
S_2 \in O(T)
\]

43
However, there are machines for which it does not hold that $S_3 \in O(T)$. In fact, $S_3$ can be exponentially large with respect to $T$. For example, there exists a successor RAM $E$ which computes $2^n$ in a time of $O(n)$. Now using the instruction $c_E$, one immediately obtains a RAM with extended instruction set which upon input of $n \in \mathbb{N}$ stores 1 in register $2^n$ and then terminates and for which $T \in O(n)$ and $S_3 \geq SB_3 \geq 2^n$.

### 4 The result

In order to formulate the main result, it is convenient to use the following function, called \textit{logarithmic function} in [1].

**Definition** For some $n \in \mathbb{N}$, we define \( l(n) := 1 \) if $n = 0$ and \( l(n) := \lfloor \log_2(n) \rfloor + 1 \) otherwise.

**Theorem** Let some RAM with extended instruction set $\Pi$ be given. Then there exists a successor RAM $\Pi'$ such that the following holds:

\( \Pi' \) terminates if and only if $\Pi$ terminates, and the output of $\Pi'$ is equal to the output of $\Pi$. Furthermore:

Let $T$ be the time complexity of $\Pi$ and $T'$ the time complexity of $\Pi'$, and let $S_1$ be the 1st space measure for $\Pi$ and $S_3'$ the 3rd space measure for $\Pi'$. Then

\[
T' \in O(T \cdot l(S_1)) \subseteq O(T \cdot l(T))
\]

and

\[
S_3' \in O(S_1 \cdot l(S_1)) .
\]

We give the proof in two parts: We first only show the result for the case that $\Pi$ is a successor RAM, and then we address the simulation of arbitrary RAMs with extended instruction set. Note that the first result is non-trivial because of the bound on the third space measure of $\Pi'$. The simulation for the first result contains the essential idea for the general result as well.

**The result for successor RAMs**

**The simulation** Let a successor RAM $\Pi$ be given. We now describe the machine $\Pi'$ used for the simulation.

Note first that – as shown in the example at the end of Section 2 – the numbers stored in the registers of $\Pi$ can be exponentially large with respect to the running
time. This is, however, not possible for successor RAMs. So we need a way to store the numbers in the registers of $\Pi$ without using too large numbers in the registers of $\Pi'$. 

A key idea for the simulation to simulate the registers and the accumulator of $\Pi$ in the following way: There are cells for data, and they always only contain 0 or 1. As a very naive approach to this idea, one might try to store the register cells in arrays. There are, however, some problems with this approach: First, how does one cope with “overflow” of arrays and second, how does one use indirect addressing in an efficient way? One possibility for the second problem would be to try to transfer such an array into one register. But note that we do not have addition instructions at our disposal, so it is unclear how to implement this idea in a sufficiently efficient way.

Rather than storing the data of one register of $\Pi$ in an array, we store it in a linked list: Each node of the list contains two entries which are each stored in one register of $\Pi'$: The first entry is a data element (being 0 or 1), and the second entry is the address of the next node.

In the same way, we simulate the accumulator of $\Pi$, and furthermore, also in this way, we implement an address register used for indirect addressing.

We use a binary tree to guarantee fast access to the simulated registers. The tree is as follows: Each node of the tree has at most two children, and the edges to the children are labeled with 0 or 1.\footnote{We only use this labeling for the present informal description of the simulation. Only the children but not the labels are stored.} Let us assume that at some time $t$, register $R_x$ of $\Pi$ contains data $d > 0$, and let $x_k \cdots x_0$ be the binary expansion of $x$ and $d_\ell \cdots d_0$ be the binary expansion of $d$. Then at the corresponding time of the simulation, there is a path from the root of the tree following the labels $x_0, \ldots, x_k$. The end of the path is the beginning of the linked list, and the data cells of the list contain $d_0, \ldots, d_\ell$. If on the other hand $d = 0$, there is no such path. (There might be a partial path in the tree but not a full path.) It is this tree structure which allows for efficient manipulation of data of $\Pi$.

During the operation, new vertices are inserted into the tree if some register is used which previously contains 0, and vertices are deleted if a register is set back to 0.

The structure just described is stored in the registers with even addresses, and one node occupies two consecutive even registers.

In order to access the storage efficiently, we use a stack and a counter. These are stored in the registers with odd addresses. Addresses of (tuples of) registers of $\Pi'$ which were used for the tree or the data cells and are deleted are put onto the
stack for reuse. (The stack is stored as an array, and each address occupies one register – as usual.) The counter stores the largest address used for the tree and the lists. If the stack is empty, the counter is incremented, and its value is used as an address.

**Illustration**  If, for example, all registers from 0 to 15 of \(\Pi\) are occupied, the tree looks like this. Here the edges with the numbers are the beginnings of the lists for the contents of the corresponding registers of \(\Pi\). (The numbers are not stored but only printed here for orientation.)

![Diagram](image)

If only registers 8, 12, 5 and 15 are occupied, the tree looks as follows:

![Diagram](image)

Again the numbers indicate that at these edges linked lists start.

If now, for example, register 15 is cleared, the corresponding edge as well as the two edges above are deleted, and the corresponding addresses of \(\Pi'\) are put on the stack for reuse.
The time and space requirements

We now outline the results on time and space requirements obtained via the simulation.

The number of registers of $\Pi'$ used for the simulation is in $O(S_1)$. Because of the use of the stack for storage management, the supremum of addresses used is in $O(S_1)$ as well. It follows that at during the whole computation the supremum of numbers stored in the registers is in $O(S_1)$.

Therefore $S'_3 \in O(S_1 \cdot l(S_1))$.

Now, in order to load the content of the simulation of register $R_i$ of $\Pi$ at time $t$ into the simulation of the accumulator or the simulated address registers we have to go along $O(l(R_i) + l(i))$ nodes of the search tree and the linked list for the register.

Again, the supremum of numbers stored in registers of $\Pi'$ is in $O(S_1)$. This implies that the logarithmic time for such an operation is in $O((l(R_i) + l(i)) \cdot l(S_1))$.

Analogous considerations apply to the computation of the successor and comparison.

All in all, the time complexity of $\Pi'$ is in $O(T \cdot l(S_1))$.

As already remarked in Section 2, $S_1 \in O(T)$ and thus $T' \in O(T \cdot l(S_1)) \subseteq O(T \cdot l(T))$.

The general case

The outline for the general case is in fact nearly as the one for the restricted case. We however also have to simulate the arithmetic instructions $c_P$, and for this we use the simulation for successor RAMs just outlined.

We simulate the storage of $\Pi'$ exactly as just described, but we only use registers $R_i$ with odd $i$. The registers $R_i$ with even $i$ are then used to simulate the higher arithmetic instructions. If $c_P$ is such a higher arithmetic instruction, then we simulate it by a successor RAM $P'$ as described in the first part of the proof.

We now outline the results on time and space complexity for the general case. In fact, with minor modifications, the analysis in the special case still applies.

The number of registers of $\Pi'$ used for the simulation of the registers, the accumulator, the stack, the counter and the address register of $\Pi$ is now in $O(SB_1)$, and the supremum of addresses used for these structures is in $O(SB_1)$. Thus the supremum of numbers stored in any register of $\Pi'$ used for these structures is in $O(SB_1)$ as well.

By the previous result, there exists a constant $C_1 > 0$ such that the supremum of addresses of $\Pi'$ and the supremum of numbers stored in the registers of $\Pi'$ used for the simulation of the arithmetic instructions is $\leq C_1 S_1$. 
All in all, we obtain $S' \in O(S_1 \cdot l(S_1))$.

The logarithmic time for the simulation of loading or storing in registers of $\Pi'$ is in $O((i(R_{i,j}) + l(i)) \cdot l(SB_1))$, and again we have an analogous result for the computation of the successor and comparison.

Furthermore, there exists a constant $C_2 > 0$ such that the simulation of any higher arithmetic instruction $C_P$ of $\Pi'$ at any particular time of $\Pi'$ can be performed in logarithmic time $\leq C_2$ times the logarithmic time of the execution of $P'$ with the particular input.

All in all, the time complexity of $\Pi'$ is in $O(T \cdot l(S_1))$.

As already remarked, $S_1 \in O(T)$. □

5 Some further remarks

We now make some further remarks related to the Theorem.

- As usual, one also can define non-deterministic RAMs with extended instruction set. There are in fact two approaches: First, one can still leave the instruction set as above (in particular, for each deterministic successor RAM $P$, we have an instruction $c_P$) but allow non-determinism in the same way as one usually does for RAM models. And second, one can in fact also extend the instruction set, allowing instructions corresponding to non-deterministic successor RAMs. In any case, the proof of the Theorem in an obvious way also leads to results on the simulation of non-deterministic RAMs.

- As a variant of this, one can consider randomized RAMs. Here the same comments as above apply. In particular, we can use the Theorem to transfer propositions on the running times of Monte Carlo or Las Vegas algorithms. Propositions concerning Las Vegas algorithms are often formulated via expected running times in the following sense: For each input $x$ the time complexity $T(x)$ is now a random variable, and one considers the function assigning to each input $x$ the expected value of $T(x)$. Propositions on expected running times defined like this can then also easily be transferred. The same applies to propositions on space complexity with respect to the various measures.

- A usual RAM type is as the successor RAM types but with two kinds of arithmetic instructions: addition and subtraction. (In [7] and [8] this is called the standard RAM.) Now, there exist $mi$-successor RAMs $A$ and $S$
which can perform addition and subtraction in linear time and with constant storage. Let now $\Pi$ be a “standard RAM” with logarithmic time measure $T$, and let the space measures $S_1, S_2, S_3$ be defined as above. Then $\Pi$ can in an obvious way be simulated by a RAM with extended instruction set $\Pi'$ such that $T' \in \Theta(T)$ and $S_i \in \Theta(S'_i)$ for $i = 1, 2, 3$. We can then apply the Theorem to simulate $\Pi$ by a successor RAM $\Pi''$ with $T'' \in O(T \cdot l(S_1))$ and $S'_i \in O(S_1 \cdot l(S_1))$. The same holds with respect to RAMs which have additional instructions for AND, OR, and XOR and/or for concatenation and shifts.

- As already mentioned, a classical and often cited computational model is the machine type defined in [1] with logarithmic cost function. In this machine type there are instructions for addition, subtraction, multiplication and division with remainder.

As shown by Schönhage ([5, Theorem 7.1]), there exists a successor RAM which computes the product of two natural numbers $m \geq n$ in a logarithmic time of $O(l(m) \cdot l(k(m)))$. Furthermore, division with remainder can be performed efficiently with Newton iteration, and the stated complexity then also holds true ([3, 4.3.3. D]).

Let now a machine $\Pi$ as in [1] be given, and let $T$ and $S_1, S_2, S_3$ be defined as above. Then by using only the second part of the simulation for the Theorem, we obtain a successor RAM $\Pi'$ simulating $\Pi$ with $T' \in O(T_u \cdot l(S_1))$ and $S'_i \in O(S_1 \cdot l(S_1))$. So we have the same conclusion as in the previous item.

- If one substitutes logarithmic by uniform time and extended logarithmic by extended uniform time, the simulation does not lead to a “quasi-optimal” result. Indeed, let $\Pi$ and $\Pi'$ be as in the simulation, and let $T_u$ and $T'_u$ be the extended uniform resp. uniform time measures. If now $\Pi$ is a successor RAM, the supremum of addresses used and the supremum of values in any register are $\leq T_u$. One then obtains $T' \in O(T_u \cdot l(T_u))$. If however $\Pi$ is some RAM with extended instruction set, one only has that the supremum of addresses used and the supremum of values in any register of $\Pi$ are $\leq 2^{T_u}$. One then merely obtains $T'_u \in O(T_u^2)$.

- Of course, with RAMs with extended instruction set and the simple uniform time measure, one can obtain nearly arbitrarily small running times. Two special cases are however worthwhile mentioning:
Let $\Pi$ be a “standard RAM” with simple uniform time measure $T_s$. Then with the simulation we obtain a successor RAM $\Pi'$ with simple uniform time measure $T'_s$ and $T'_s \in O(T^2_s)$. The argument for this is exactly as the one for the previous item. This result is given in [8, Theorem 19.28].

However, if one allows all four arithmetic instructions, one obtains a dramatically different model; see [2], [4] and [8, Theorem 20.12], [8, Theorem 20.35]: The set of languages which can be recognized in polynomially bounded time on a nondeterministic machine can then also be recognized in polynomially bounded time on a deterministic machine and is equal to the set of languages which can be recognized in polynomially bounded space on a Turing machine. From a complexity theoretic point of view, this model can be considered as a parallel model.

- One can “iterate” the definition of the machine type “RAM with extended instruction” set by defining a new type which has as arithmetic all instructions of the RAM with extended instruction set. By iterating this procedure, we obtain a sequence RAM types indexed by the natural numbers; let us call any machine of these types a RAM with iteratively extended instruction set. We can now also iterate the definition of the extended logarithmic time measure and the space measures, obtaining in this way measures for all these machines. Let now such a RAM $\Pi$ be given. Then one can also apply the simulation iteratively. Finally, one obtains a successor RAM $\Pi'$ which simulates $\Pi$ such that the following holds: With the notations as in the Theorem and the usual $\tilde{O}$-notation to capture logarithmic factors, we have:

$$T'_s \in T \cdot Poly(l(S_1)) \subseteq \tilde{O}(T) \text{ and } S'_s \in \tilde{O}(S_1)$$

- It would be very interesting to have a general “quasi-optimal” result on the simulation of random access machines in some model by Turing machines. However, no such result is known. The following statement is however obvious: Let $\Pi$ be a RAM with extended instruction set. Then there exists a Turing machine (with 1-dimensional tapes) simulating $\Pi$ with a time complexity of $O(T \cdot S_1) \subseteq O(T^2)$.

6 Summary

We give a summary of the definitions and results of this work on an intuitive level.
The Bulletin of the EATCS

The starting point of this work is the observation that often the analysis of algorithms is performed in an ad-hoc way without reference to a specific model of computation. Implicitly however, the algorithms are usually analyzed in some kind of random access machine (RAM) model with some kind of instruction set. This motivates the search for a general transfer result to a truly bit-oriented model of computation. Such a result is given in this work.

Briefly, the result can be stated as follows: If one defines the time and space requirements of the instructions of the model in a bit-oriented way, one can obtain a transfer which is “quasi-optimal”, i.e. “optimal up to a logarithmic factor”.

Generally speaking, the result shows that if one employs the usual $\tilde{O}$ or $O^*$ notation, it really is justified to take an intuitive and not too formal approach to complexity of algorithms.

Two aspects should however be added to caution the reader:

First, if one uses the $O$-notation and gives explicit “logarithmic terms”, it really is necessary to first state the corresponding computational model. (At least as long as no stronger simulation result is known.)

Second, one might argue that a more adequate model of computation for algorithms with large space requirements is the multitape Turing model. There is, however, no general “quasi-optimal” transfer result from RAM models to the Turing model known.

Acknowledgment

I thank Pierrick Gaudry for discussions on computational models.

References


THE CONCURRENCY COLUMN

BY

LUCA ACETO

School of Computer Science, Reykjavik University
Kringlan 1, 103 Reykjavik, Iceland
luca@ru.is, http://www.ru.is/faculty/luca

This installment of the concurrency column is devoted to a piece by Vasco Thudichum Vasconcelos on the topic of session types. Vasco has been one of the original proposers of the notion of session type in order to specify the allowed interactions between two parallel processes communicating by means of message passing. Since that early work, the notion of session type has proved to be very flexible and fruitful. The literature on session types is already very substantial and is growing fast. I hope that Vasco’s contribution to the Concurrency Column will prove useful to researchers who are interested in understanding the ideas behind session types and their range of possible applications.
SESSIONS, FROM TYPES TO PROGRAMMING LANGUAGES

Vasco T. Vasconcelos
LaSIGE, Faculty of Sciences, University of Lisbon

Abstract

We discuss session types independently of any programming language. We then embody the notion in languages from three different paradigms: the pi calculus, a functional language, and an object-oriented language.

1 Introduction

Session types allow a concise description of a continuum of interactions among different partners. The notion was originally introduced with the aim of specifying and disciplining interactions between two partners running in parallel and communicating via message passing [8, 12]; the setting was then a mild variation of the pi-calculus [10]. Since then, session types were incorporated in different paradigms, including functional languages, object-oriented languages and service-oriented computing, to name a few. Reference [2] presents a recent overview of the field. What once looked like a notion of types tuned for a particular paradigm of computation, turned out to be a quite rich, language-independent idea.

Traditionally, session types have been used to describe linear interactions only, that is, interactions between exactly two partners (or two threads). But linearity alone is not enough to express the rich computational structures one might be interested in. As such, (linear) session types are often complemented with some other sort of (shared) types, resulting in two disjoint categories for types incorporating a certain dose of redundancy. Inspired on a formulation of linear types for functional programming [15], where types (pre-types, more precisely) are annotated with qualifiers, we have formulated an elegant theory of session types for the pi-calculus [14]. A pre-type equipped with a linear qualifier plays the role of a traditional session type; when annotated with an unrestricted qualifier, the type describes an entity shared by multiple partners.

In the vast majority of applications, sessions types are associated to message passing, describing the messages flowing on communication channels, regardless
of the host programming language (pi calculus, functional, or object-oriented). But this need not be the case. Session types have been used to describe the behavior of objects in component models [13], as well as attached to class definitions where they specify possible sequences of method calls [5].

This paper presents a version of session types equipped with \texttt{lin/un} qualifiers [14], meant as descriptions for \textit{communication media} in general. It then incorporates the notion into three different programming paradigms: a purely concurrent message passing system (embodied by the pi calculus), a multithreaded functional language equipped with message passing, and a multithreaded object-based language. The communication medium is instantiated as communication channels in the first two languages; and as object references in the last. The tone is left informal, references point to detailed descriptions.

2 Session Types

This section introduces the notion of \texttt{lin/un} qualified session types as well as our running example.

The running example. Our running example is that of an online petition service. Petition writers start by providing the title of the petition, a piece of text describing the situation and what is needed, and the deadline for signature collection. In order to make life easier to petition writers, the service allows this information to be added with no particular order; writers can even resubmit information if needed.

Once writers are happy with the provided petition details, time comes for submission. This is the point where writers commit to the uploaded data and seek approval for starting the petition. If accepted, meaning for example that the deadline for signature collection is not in the past or that there is no other running petition under the same name, then writers may start promoting the petition. Promoting a petition means two things: signing and disseminating. The petition writer may sign the petition, and must get people to sign it, by letting them know of the newly created petition. Hopefully the writer’s acquaintances will further sign and disseminate the petition, thus contributing to the success of the campaign.

We formalise the protocol that runs amongst the petition service, the petition writer, and the signatories. The protocol can be divided in two phases—\texttt{setup} and \texttt{promotion}—separated by a submit operation. During the setup phase, if an unbounded number of data (titles, descriptions, and dates) is reaching the server by an arbitrary order, then it must be tagged, for otherwise servers would face difficulties in distinguishing, say, two consecutive titles from a description followed by a title. Therefore writers start by selecting one particular operation:
setTitle for setting the title, setDate for setting the deadline, submit for moving to the promotion phase, or even setting the description of the situation (omitted henceforth). Selecting one particular operation is described by a type of the form ⊕{setTitle: ..., setDate: ..., submit: ...}, where the dots replace the protocol description after the selection operation.

Following a setTitle operation, writers are supposed to send the actual title (a string); similarly a date is supposed to follow a setDate selection. Output of a string is denoted by !string, so that our protocol now looks like this: ⊕{setTitle: !string..., setDate: !date..., submit: ...}. After successful uploading the title or the date, writers are given the chance to revise their data or to submit the proposal, so that the protocol returns to the starting point. This requires a recursive type, which we write in the form of an equation.¹

Petition = ⊕{setTitle: !string.Petition, setDate: !date.Petition, submit: &{accepted: Promotion, denied: ?string.end}}

As discussed above, a submission operation may result in acceptance or rejection. We could model such an outcome with a boolean value, but the ensuing protocol crucially depends on this outcome: move to the promotion phase or halt. As such the writer must expect the selection of an operation—accepted or denied—coming from the server. Dually, the writer must offer the server a menu composed of the two operations; we write such a type as &{accepted: ..., denied: ...}.

If denied then the writer receives a reason in the form of a string and the protocol terminates (the writer may try again, but on a different run of the protocol). If accepted then the protocol moves to the promotion phase. Input is denoted by ?, so that receiving a string becomes ?string; we represent protocol termination by end. Here is our type so far:

Petition = ⊕{setTitle: !string.Petition, setDate: !date.Petition, submit: &{accepted: Promotion, denied: ?string.end}}

During the promotion phase all one can do is to sign the petition by sending a signature (in the form of a string), so that we have

Promotion = !string.Promotion

We have hinted above that the protocol is to be run amongst the petition service, the petition writer, and the signatories. If we assume that no two consecutive operations (input, output, branch, or select) in a protocol are atomic by default, then some care must be exerted on how many partners may know the protocol medium at each time. If this medium is disseminated in the setup phase, then

¹The attentive reader certainly noticed that writers may submit without first uploading the petition details. We assume that there is a default title ("Save me") and a default deadline (1/1/1970). We leave it as an exercise to refine the type in such a way that a) the submit button is pressed only after the setTitle and the setDate have been pressed at least once each, and b) each of these two last buttons may be pressed more than once.
race conditions may arise when, e.g., the server receives a commit operation from one partner, followed by a setDate from a different, unsynchronized partner. On the other hand, during the promotion phase, the success of the petition crucially depends on dissemination, so that we want the protocol medium shared by an unbounded number of potential promoters (signatories and disseminators).

Towards this end we qualify each operation in a type with one of two qualifiers: \texttt{lin} denotes a linear operation; \texttt{un} denotes an unrestricted, or shared, operation. When the protocol is in a \texttt{lin} state then the programming language must guarantee that exactly two partners (server and writer) know the protocol medium; when in an \texttt{un} state then an unbounded number of partners (potentially zero) may have access to the medium. Our fully qualified type is thus:

\begin{verbatim}
Petition = \texttt{lin}@{setTitle: \texttt{lin}!string.Petition, setDate: \texttt{lin}!date.Petition, submit: \texttt{lin}/[accepted: Promotion, denied: \texttt{lin}?string.lin end]}

Promotion = \texttt{un}!string.Promotion
\end{verbatim}

There is one last question that we must answer. How do petition writers and petition servers initiate a particular run of the protocol? Petition servers are usually installed on well-known names. It is on one such name that writers and servers agree on initiating a new run of the petition protocol. The protocol is itself started from a small bootstrap protocol, where the server provides the writer with a fresh Petition. If the only thing the server does is to start new Petition protocols, then its type is \texttt{Server} = \texttt{un}!Petition. which we abbreviate to \texttt{*!Petition}.

So far we have been looking at the protocol from the point of view of writers and promoters. How do things look like when seen from the server side? In order to comply with the writer’s expectations, servers must start by offering a menu composed of operations \texttt{setTitle}, \texttt{setDate}, and \texttt{submit}, which we write as \texttt{lin}&{setTitle: \texttt{...}, setDate: \texttt{...}, submit: \texttt{...}}. After a \texttt{setTitle} operation the server must input a string (?string); after \texttt{setDate}, it is time to input a date (?date). After \texttt{submit}, the server must select one of the two operations—\texttt{accepted} or \texttt{denied}—on the client, which we write as \texttt{lin}@{accepted: \texttt{...}, denied: \texttt{...}}. When \texttt{denied}, then the service must output a string and terminate with \texttt{lin end}; the server and the client terminate the protocol together. It should by now be clear that, in order for communication to run smoothly among the various partners involved, when one says output (!), the other says input (?), when one says select (@), the other says branch (&), and when one says terminate (end) so does the other. The \texttt{un}/\texttt{lin} qualifiers must match in each case. The types constructed in this way are said to be \textsf{dual}.

**Session types.** The types that describe our protocols are generated by the grammar in Figure 1, where we use letter \texttt{p} to denote an unqualified (or pre-) type, and letter \texttt{T} to describe a type. Recursive types are required to be contractive, that is,
BEATCS no 103

THE EATCS COLUMNS

Figure 1: The syntax of types

\[
q ::= \text{Qualifiers:} \quad \text{!}T.T \quad \text{send} \\
\text{lin} \quad \text{linear} \quad \ominus \{ l_i : T_i \}_{i \in I} \quad \text{select} \\
\text{un} \quad \text{unrestricted} \quad \& \{ l_i : T_i \}_{i \in I} \quad \text{branch}
\]

\[
p ::= \text{Pretypes:} \quad T ::= \text{Types:} \\
\text{unit} \quad \text{unit} \quad q \quad p \quad \text{qualified pretype} \\
\text{end} \quad \text{termination} \quad a \quad \text{type variable} \\
?T.T \quad \text{receive} \quad \mu a.T \quad \text{recursive type}
\]

Figure 2: The dual function on types

\[
q ?T.U = q !T.U \quad q !T.U = q ?T.U \quad q \text{end} = q \text{end} \\
q \ominus \{ l_i : T_i \}_{i \in I} = q \& \{ l_i : T_i \}_{i \in I} \quad q \& \{ l_i : T_i \}_{i \in I} = q \ominus \{ l_i : T_i \}_{i \in I} \\
\mu a.T = \mu a.\overline{T} \quad \overline{a} = a
\]

containing no subexpression of the form \( \mu a_1 \ldots \mu a_n.a \). The equations introduced above are transformed into recursive types in the standard way:

\[
\text{Petition} = \text{rec a.lin}@[\text{setTitle: } \text{string.a, setDate: } \text{date.a, submit: } \ldots]
\]

In the presence of recursive types, we define type equality as the equality of the regular infinite trees obtained by the infinite unfolding of recursive types. The formal definition, which we omit, is co-inductive. In this way we can use types \( \text{un!string.rec a.un!string.a} \) and \( \text{rec a.un!string.un!string.a} \) interchangeably, in any mathematical context. This allows us never to consider a type \( \mu a.T \) explicitly (or \( a \) for that matter). Instead, we pick another type in the same equivalence class, namely the type obtained by replacing in \( T \) occurrences of type variable \( a \) by type \( \mu a.T \), usually written \( T[\mu a.T/a] \). If the result of the process turns out to start with \( \mu \), we repeat the procedure. Contractiveness ensures the termination of the unfolding process. In other words, we take an equi-recursive view of types [11].

Rather than providing a co-inductive definition of duality, we start by defining a function from types to types as in Figure 2. Then, to check that a given type \( T_1 \) is dual of another type \( T_2 \), we first build the dual of \( T_1 \) and then check that the thus obtained type is equivalent to \( T_2 \). For example, to show that type \( \text{rec a.un!string.un!string.a} \) is dual to \( \text{rec b.un?string.b} \), we first build type \( \text{rec a.un?string.un?string.a} \), dual of the former and then check that it is equivalent
The Bulletin of the EATCS

Context split

\[
\begin{align*}
\emptyset &= \emptyset \circ \emptyset \\
\Gamma &= \Gamma_1 \circ \Gamma_2 \\
\Gamma, x : \text{un} p &= (\Gamma_1, x : \text{un} p) \circ (\Gamma_2, x : \text{un} p) \\
\Gamma &= \Gamma_1 \circ \Gamma_2 \\
\Gamma, x : \text{lin} p &= (\Gamma_1, x : \text{lin} p) \circ \Gamma_2 \\
\Gamma &= \Gamma_1 \circ \Gamma_2
\end{align*}
\]

Context update

\[
\begin{align*}
\Gamma &= \Gamma + \emptyset \\
\Gamma, x : T &= \Gamma_1 + (\Gamma_2, x : T) \\
\Gamma, x : \text{un} p &= (\Gamma_1, x : \text{un} p) + (\Gamma_2, x : \text{un} p)
\end{align*}
\]

Figure 3: Context split and context update

to the latter. Duality is defined on session types only; it does not apply to the unit type. Would we require boolean or function types, say, duality would not be defined on them either.

Programs are usually typed against a context describing the types for the free identifiers. Typing contexts are finite maps \( \Gamma \) from identifiers (or variables, denoted by \( x \)) to types. Symbol \( \emptyset \) indicates an empty map. Given an arbitrary map \( \Gamma \) and a variable not in the domain of \( \Gamma \), we denote by \( \Gamma, x : T \) the map equal to \( \Gamma \) everywhere except at \( x \) where \( \Gamma(x) = T \). We maintain the linearity invariant through the standard linear context splitting operation. When type checking processes with two sub-processes we pass the unrestricted part of the context to both processes, while splitting the linear part in two and passing a different part to each process. Figure 3 defines the context splitting relation \( \Gamma = \Gamma_1 \circ \Gamma_2 \). Notice that in the third rule, \( x \) is not in \( \Gamma_2 \) otherwise it would be in \( \Gamma = \Gamma_1 \circ \Gamma_2 \) and the result \( \Gamma, x : \text{lin} p \) would not be defined, and similarly for the last rule and \( \Gamma_1 \).

Unlike conventional linear values that are consumed once they become used, values that describe the medium on which protocols run are consumed piecewise: an input on a medium of type \( q?T_1.T_2 \) renders the same medium at type \( T_2 \). We introduce a context update operation for the effect (Figure 3). If \( q \) is \( \text{lin} \) then, by virtue of context splitting, reading the medium removes its type \( q?T_1.T_2 \) from the context, while context update adds the continuation type \( T_2 \) (second rule). If, on the other hand, \( q \) is \( \text{un} \) then the type remains in the context, and we must add a type \( T_2 \) equivalent to \( q?T_1.T_2 \), according to the definition of context update (third rule). Since we want to add the continuation \( T_2 \), it must be the case that \( T_2 \) is equivalent to \( \text{un}T_1.T_2 \), which can happen if, e.g., \( T_2 \) is of the form \( \mu a.\text{un}?T_1.a \). This form of type is so common that we introduce an abbreviation for it, \( *?T_1 \), as we have seen.

59
Linear type systems follow an invariant whereby unrestricted data structures may not include linear data structures. This is usually accomplished by defining two predicates \( \text{un} \) and \( \text{lin} \) that operate both on types and on contexts. The rules state that linear data structures can hold objects of a linear or unrestricted nature, but that unrestricted data structures can only contain unrestricted values. Making \( q \sqsubseteq q' \) the smallest reflexive such that \( \text{lin} \sqsubseteq \text{un} \) we define \[15\]:

- \( q(T) \) when \( T = q' p \) and \( q \sqsubseteq q' \)
- \( q(\Gamma) \) when \( x : T \in \Gamma \) implies \( q(T) \)

Notice that in particular \( \text{lin}(T) \) is true for any \( T \), and similarly for contexts.

3 Session types in the pi-calculus

We now embody the types as described in the previous section in a message passing concurrent language, the pi-calculus \[10\]. Our medium of communication (as we put it in the previous section) is channels where messages flow. A channel at a linear type is held by exactly two threads; a channel at an unrestricted type is held by zero or more threads.

The running example in the pi calculus. We assume a petition server installed at the well-known channel \( ps \). Petition writers read from this channel a petition channel \( p \) (line 3, Figure 4). Our writer starts by setting the deadline, then the title\(^2\), and finally decides to adjust the deadline (lines 4–6). Once happy with the information provided, the writer submits the request, and waits for an answer (lines 7–8). If accepted, then the writer distributes the petition channel to its two acquaintances (Signatory1 and Signatory2), and signs the petition himself (lines 9–11); if denied then the writer receives the reason for denial (in the form of a string), closes the channel end point and terminates (lines 13–14).

The three acquaintances have different behaviours. The first gets channel \( p \) and signs the petition (lines 17–18); the second further sends the petition to another potential signatory (Signatory3) and signs (line 21). Finally, Signatory3 gets the channel but decides neither to sign nor to further disseminate (lines 23–24).

Let us have a look at the server side. Our implementation is divided into three components: the Server itself that creates and sends new channels on \( ps \); the Setup that gathers the information on a particular petition, and the Promotion that collects the various signatures. Petition servers must provide (on well-known name \( ps \)) fresh linear petition channels. This is accomplished in the pi-calculus with a \texttt{new}-constructor that creates a fresh channel, different from all others (line 3). One end

\(^2\)“Save the Iberian wolf”, \textit{Canis lupus signatus}. 
The Bulletin of the EATCS

1 SaveTheWolf :: ∗?Petition
2 SaveTheWolf ps =
3 ps?p.
4 p◁setTitle. p!?"Save the Wolf".
5 p◁setDate. p!(31,12,2100).
6 p▷submit. ps
7 p▷close ps
8 p▷{ accepted:
9 Signatory1 p |
10 Signatory2 p |
11 !"me"
12 denied:
13 p?x.
14 close p
15 }.
16 Signatory1 :: ∗string
17 Signatory1 p =
18 !"signatory1"
19 Signatory2 :: ∗string
20 Signatory2 p =
21 Signatory3 p | !"signatory2"
22 Signatory3 :: ∗string
23 Signatory3 p =
24 inaction

Figure 4: Petition example in the pi-calculus

of this channel, denoted by p2, is passed to potential writers (line 4); the other end, called p1, is passed to process Setup, together with the default deadline and title (line 5).

Process Setup receives the petition channel, the default deadline, and the default title, and interactively updates the last two (lines 9–10). Our simplistic server accepts each single petition (line 11). The protocol now passes to the promotion phase, by providing the Promotion process with the petition channel p and an empty list, where the signatory names are to be stored. In order to simplify the example, we use a data type for lists, where [] denotes the empty list and s :: l denotes a list composed of an element s at the head and a list l at the tail. Such a data type would have to be encoded in the base language [9, page 106]. Process Promotion receives a signature s on channel p, stores it in the list (s :: l) and recurs.3

3Notice that, in the example, we use symbol :: both as list concatenation and to introduce types in processes.
Typing rules for values

\[
\begin{align*}
\text{un}(\Gamma) & \quad \Gamma \vdash (): \text{unit} \\
\end{align*}
\]

Typing rules for processes

\[
\begin{align*}
\text{un}(\Gamma) & \quad \Gamma \vdash x: \text{lin} \quad \Gamma_1 \vdash P_1 \quad \Gamma_2 \vdash P_2 \quad \Gamma, x: T \vdash y: T \vdash P \\
\text{inaction} & \quad \Gamma \vdash \text{close} x \\
\text{lin} \rightarrow \text{lin} \quad & \quad \Gamma_1 \circ \Gamma_2 \vdash P_1 | P_2 \\
\text{lin} \rightarrow \text{res} \quad & \quad \Gamma, x: T \vdash y: T \vdash P \\
\text{lin} \rightarrow \text{lin} \quad & \quad \Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x!v; P \\
\text{lin} \rightarrow \text{res} \quad & \quad \Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash x?v; P \\
\text{lin} \rightarrow \text{res} \quad & \quad \Gamma_1 \circ \Gamma_2 \circ \Gamma_3 \vdash q(x); P \\
\end{align*}
\]

Figure 5: Typing rules for the pi-calculus

Our Main process creates a channel and distributes one of its ends (ps1) to process Server and the other end (ps2) to the petition writer, SaveTheWolf. In the previous section we discussed the type of the well-known name where new petitions are to be requested: ∗! Petition or ∗? Petition, depending on the point of view; these are the types of the two ends of the newly created channel, ps1 and ps2, respectively.

For the client, channel ps carries p typed at type Petition. The initial, linear, part of the channel is consumed in lines 3–8 of process SaveTheWales. When control reaches line 9, channel p is of type ∗! string, allowing it to be freely passed around and used for signing (lines 9-11, as well as processes Signatory1, Signatory2, and Signatory3). On the server side, the initial linear part of the type dual of to Petition is consumed in process Setup, whereas the unrestricted part (∗? string) is used in process Promotion.

Typing pi processes. The typing rules for the pi-calculus are in Figure 5. We omit the grammar of values and processes, which can be easily inferred from the rules. Typing judgements for values are of the form \( \Gamma \vdash v: T \), indicating that value \( v \) has type \( T \) under context \( \Gamma \), as usual. Judgements for processes are of the form \( \Gamma \vdash P \), testifying that process \( P \) is well typed under context \( \Gamma \).

We briefly comment on the rules. The rules for values and for process inaction make sure that the unused typing context \( \Gamma \) is unrestricted, thus ensuring that linear values are completely consumed. The rule for closing channel ends, T-Close, requires a channel ready to be closed: linear (no other process may know it) and
at end (the protocol on the channel is completed). The rule for parallel composition, T-Pα, splits the incoming context in two and passes each part to a different process. The rule for name restriction, T-Res, simply adds two dual types to the context, each will type one end of the newly introduced channel. The rule for output processes of the form \( x!P \), T-Out, splits the context in three parts, one to type each of the constituents of the process. The continuation \( P \) is typed at context \( \Gamma_3 \) updated with \( x \) at the continuation type \( T_2 \). This means that either \( q = \text{lin} \), and hence \( x \) is not in \( \Gamma_3 \), or \( q = \text{un} \) and in this case \( T_2 \) must be equal to \( \text{un}!T_1.T_2 \).

The last rule in the figure, T-In, accounts for both simple (linear) or replicated (unrestricted) input. In the pi-calculus one traditionally indicates a perennial process by prefixing it with an exclamation mark. Since such a symbol was taken in our language to designate output, we then take the chance to align the syntax of the language with that of types. We write an ephemeral input process as \( \text{lin}x?y.P \) and a persistent (replicated) input process as \( \text{un}x?y.P \). The \( q(\Gamma_2) \) proviso in the rule for input makes sure that the variables in the body of a replicated process are unrestricted themselves.

In the presence of free output it is well-known [16] that we must differentiate the two ends of a channel. Several techniques are known: annotating channel ends with distinct, +, −, polarities [6], using type constructors that can talk about the ends of a same channel [7], or work with a double binder that binds together distinct identifiers for the two ends of a same channel [14]. Here we follow the last method.

We have lived two decades without requiring a close operation for the pi-calculus. Why now? The truth is we do not strictly need it. Session types that start linearly may terminate as \( \text{un} \) end, but then the runtime has to run a potentially expensive garbage collection procedure in order to deallocate the data-structures necessary to implement channel operations. On the other hand, session types that terminate in \( \text{lin} \) end allow a close operation to explicitly deallocate the supporting data-structures. Channels of type \( \text{un} \) end, even though they cannot be used for communication, can still be freely passed around (and stored in data structures) but cannot be deallocated for there is an undetermined number of references to it.

We leave as an exercise the design of the rules for the branch and the selection processes (the interested reader may want to check a solution in reference [14]).

**Back to the example.** The code in Figure 4 omits the \( \text{lin} \) qualifier in input prefixes. The \( \text{un} \) input prefix qualifier cannot be found for we have used recursive definitions in place of replication. The encoding of recursion into replication is standard [9, page 94]. For example, the Promotion definition in lines 15–16 of the server is transformed into \( \text{un} \) promotion?\( p.l \). \( \text{lin} \) \( p?\)s.promotion\( \langle p.s::l \rangle \), whereas the call in line 12 becomes promotion\( \langle p.[] \rangle \). Finally, a (new promotion) binder as
well as the replicated process itself must be placed at, say, the top level process, line 2 of Main.

In the example we use polyadic messages, as opposed to the monadic messages proposed in the typing rules where messages can carry exactly one value. This happens, for example, in the expansion of the Promotion process described above. Once again the polyadic-to-monadic encoding is well-known [9, page 93] (cf. [14]). To atomically send the two arguments, p and [], on channel promotion, we translate message promotion!(p,[]) into a process that implements a simple protocol: \((new\ c1,c2)\) promotion!c2.\(c1!p.\) close c1. Such a process creates a new channel; one end, c2, is sent to the client, the other, c1, is used to transmit the two arguments. On the other hand, the receiving side, un promotion?c.\(\lin\ c?p.\lin\ c?!p.\) close c.\(P\) is translated into process un promotion?c.\(\lin\ c?p.\lin\ c?!p.\) close c.\(P\) that receives a fresh linear channel on which the two parameters may be received without risk of interference.

The type of channel promotion can be precisely captured by our types. Because there is a replicated receptor installed at the channel, the type takes the form \(\ast\ ?T\), as seen from the point of view of the receiver. Type T describes the little (linear) protocol used to receive the two parameters, namely \(\lin\ ?(\ast\ ?\text{string}).\lin\ ?\text{stringList}.\lin\ \text{end}\).

4 Session types in a functional language

We now address the design of a call-by-value functional multi-threaded programming language. We add to a linear functional programming language [15] a notion of channels, akin to that described in the previous section for the pi-calculus. In addition to the lambda-calculus constructors—basic values, variables, abstraction, application and pairs—we rely on operations for channel creation, sending/receiving/branching on a channel, as well as for forking new threads. For the new constructs, we stick as much as possible to the syntax of the previous section.

The running example in a functional language. Because we use the same syntax for channel operations, the code for the client, Figure 6, should be easy to follow. The main difference with respect to the pi-calculus code is that, once (and if) the petition request is accepted by the server, the writer forks two threads, one for each signer (lines 9–10). Rather than sending p on a channel (known to the writer and to a signer), the channel is passed as a parameter to the function. To align our language with the expectations imposed by functional programming, we allow writing \(x?\) to receive a value on a channel \(x\), without explicitly mentioning the variable that will hold the value, nor the term that constitutes the continuation. In our example, the petition writer simply discards the reason for denial (line 14).
Typing functional terms. We need one more type for functions; more precisely one pretype \( p \to p \), which we add to those in Figure 1. As discussed in Section 2, duality is not defined on this type. Figure 7 presents the typing rules for the language. Once again, apologising for the inconvenience, rather than presenting the syntax we ask the reader to read it from the terms in the conclusion of the rules. Typing judgements are of the form \( \Gamma_1 \vdash M : T ; \Gamma_2 \) conveying the idea that term \( M \) has type \( T \) under context \( \Gamma_1 \). The “continuation” context \( \Gamma_2 \) describes the residual types of the variables used in \( M \) for channel operations (input, output, receive, select). If \( T \) is a type of the form \( \text{lin} \text{un unit} T' \), we have:

\[
\begin{align*}
  x : T &\vdash x? : \text{un unit} \quad x : T' \\
  x : T &\not\vdash () : \text{un unit} \quad x : T
\end{align*}
\]

The main challenge in the design of a type system for a functional language...
that is a system for the pi-calculus (Figure 5). In any case, functions must consume their linear resources (unlike the system in reference [5]), as enforced by the qualifier of the function is unrestricted (cf. rules T-I and T-Out) works as an elimination rule for type q?(Γ_1,Γ_2), where the type of expression q? alone. In our language, the continuation, if present at all, comes in the form of an abstraction or of the second component of a pair, and we would not like to mention them explicitly in the typing rules. We solve the problem by adding an extra context at the right hand side of judgements.

Equipped with such judgements, the rule for the input expression (T-In) acts effectively as an elimination rule for type q?Γ_1,Γ_2, where the type of expression x? is Γ_1 and the context available to the continuation contains x at type Γ_2. Similarly, the rule for the output process (T-Out) works as an elimination rule for type q!Γ_1,Γ_2, where the type of expression x! is understood as a (linear) function receiving Γ_1 (and delivering un unit) and where the continuation sees x at type Γ_2. In both rules the context available to the continuation is Γ_2 + x: Γ_2, thus ensuring that, when in presence of an unrestricted Γ_2 (occurring both in Γ_1 and Γ_3), the type of x available to the continuation is equal to the initial type (un?Γ_1,Γ_2 or un?Γ_1,Γ_2), thus ensuring that, when in presence of an unrestricted Γ_2, the type of x available to the continuation is equal to the initial type (un?Γ_1,Γ_2 or un?Γ_1,Γ_2), that is Γ_2 is, e.g., of the form μa.un?Γ_1,a (cf. rules T-In and T-Out in the type system for the pi-calculus, Figure 5).

In the rule for abstraction (T-Abs), if the qualifier of the function is un (meaning that the function can be used multiple times), then all free variables in the body of the function (hence in Γ_1) must equally be of unrestricted types (cf. the situation of the replicated input in Figure 5). In any case, functions must consume all their linear resources (unlike the system in reference [5]), as enforced by the

\[
\begin{align*}
\Gamma_1 + x: q?T_1,T_2; \emptyset & \quad \text{un}(\Gamma_2) \\
\Gamma_1 \circ \Gamma_2 + x? : T_1; \Gamma_2 + x: T_2 & \quad \text{un}(\Gamma_2) \\
\Gamma_1 + x: q!T_1,T_2; \emptyset & \quad \text{un}(\Gamma_2) \\
\Gamma_1 \circ \Gamma_2 + x!: \text{lin}(T_1 \to \text{un unit}); \Gamma_2 + x: T_2 & \quad \text{un}(\Gamma_2)
\end{align*}
\]

Figure 7: Typing rules for the functional language
proviso \( \text{un}(\Gamma_2) \). Being values, functions leave no linear values to be consumed by the continuation, as witnessed by the final empty context.\(^4\) The rule for function application (T-App) splits the incoming context in two parts, \( \Gamma_1 \) and \( \Gamma_2 \), made available to its two parts \( M_1 \) and \( M_2 \). If function \( M_1 \) leaves linear values \( \Gamma_3 \) to be consumed by the continuation (as it happens in case \( x! \)), these are added to those of the argument, \( \Gamma_4 \), and made available to the continuation of the application.

What we do not know is how to type a general input or output expression, \( M! \) or \( M? \), for we need to get hold of the channel reference \( x \) in order to advance its type. For both cases we provide a let construct: rather than writing \( M! \), we write \( \text{let} \ x = M \ \text{in} \ x! \), and in place of \( M? \), we write \( \text{let} \ x = M \ \text{in} \ x? \). Very much like rule T-App, rule T-LET splits the incoming context in two parts, one for each subterm. The difference is that the linear values not completely consumed by \( M_1 \) (present in \( \Gamma_3 \)) are added to \( \Gamma_2 \), in order to type the body \( M_2 \), whereas in the case of rule T-App they are added directly to the final context. The linear values not completely consumed by \( M_2 \) (in \( \Gamma_4 \)) are those provided to the continuation of the let. In the same vein, we cannot directly type term \((x!)(x?)\) since we would be typing subterm \( x! \) first where we expect subterm \( x? \) to be evaluated first. Once again, the let-construct allows to make explicit the evaluation order, \( \text{let} \ y = x? \ \text{in} \ (x!)y \).

The fork construct can be intuitively described by a type of the form \( \text{un}(\text{un}\ p \to \text{un}\ \text{unit}) \), more precisely by a type schema, ranging over all pre-types \( p \). The \( \text{un}(\Gamma_2) \) proviso together with the fork \( M \) unrestricted type, \( \text{un}\ p \), makes sure that the forked expression \( M \) consumes all its linear resources.

Semicolon is as usual a derived construct. But we must use let, rather than abstraction. Expression \( (;x?) \) cannot be translated as \((p\_x?)()\) for it would not be typable under context \( x: \text{lin}\!?\text{unit}.\text{lin} \ p \) (the function does not consume all its linear resources). Instead we use \( \text{let} \ _ = (_) \ \text{in} \ x? \) which allows to leave \( x \) at type \( \text{lin} \ p \) to the continuation.

We have seen that functions must consume all their linear resources. This means that we cannot type a function of the form \( \lambda x. x!5 \) with \( x \) of the type \( \text{lin}?\text{unit}.\text{lin} \ p \). The alternative is for the function to return all its unused linear resources, as in \( \lambda x. (x!5; x) \) which can be typed as \( \text{lin}(\text{lin}?\text{nat}.\text{lin} \ p \to \text{lin} \ p) \). Similarly, rather than writing \( \lambda x. x? \), we must write \( \lambda x.(x?, x) \) which can be typed as \( \text{lin(lin}\!?\text{nat}.\text{lin} \ p \to (\text{nat} \to \text{lin} \ p)) \), following the approach described in reference [4].

We leave as an exercise deriving the typing rules for pair construction and deconstruction; they can be easily adapted from reference [15, page 10], as well as adapting the rule for closing channel ends. We equally leave as an exercise the rules for branch and selection. The reader may as well consider adding a fixed point operator to the language, without which the server in Figure 6 would not be typable.

\(^4\)Context \( \Gamma_2 \) would as well do since \( \Gamma_1 + \Gamma_2 = \Gamma_1 \) when \( \text{un}(\Gamma_2) \) and \( \Gamma_1 \circ \Gamma_2 \) defined.
Back to the example. Following our convention, we have omitted all unqualifiers in code in Figure 6. The functions that compose the client are used only once, they could easily be typed at a list type as well. Those for the server are recursive; they must be unrestricted. In order to comply to the restriction on function calls, whereby a channel cannot be used both in the function and in the argument, code must be adapted. For example, the function call on server, line 18, becomes let \( s = p? \) in promotion \( p \ (s::l) \).

5 Session types in an object-based language

We now incorporate session types in a conventional class-based imperative programming language. The language features multithreaded concurrency, where different threads communicate solely by calling methods in remote objects. We thus see that the communication medium, identified in Section 2 and embodied as communication channels in the pi calculus and in our functional language (Sections 3 and 4), is instantiated here as object references. Whereas in channel-based languages, processes communicate by exchanging messages on (session governed) channels, in our object-oriented language threads communicate solely by calling methods on (session governed) object references. This option clearly contrasts with more conventional approaches that add communication channels to an object-oriented language (e.g., [3, 5]).

The running example in a language of objects. The code for our running example can be found in Figure 8. Composed of four classes: SaveTheWolf, PetitionServer, Signatory, and Main, we try to follow as close as possible the architecture of the solutions in previous sections.

While we could force, via a suitable encoding, the Petition session type as identified in Session 2 into our language, we seek a natural incorporation of the protocol into familiar OO concepts. With respect to session types, method calls bring some restrictions as well facilities. A selection operation (previously identified with a left triangle, \( \triangledown \)) can be identified with a method call, while an output operation can only be identified with argument passing within a method call. An input operation can only occur as the result of a method call. What about branching? How can a target object force a branch on a client? For a simple binary branch, we could stipulate that boolean methods would force such a test, via conditional expressions. For more general branching structures we use conventional enumerations (enum) and a switch construct.

In object-oriented languages, method call and argument passing are usually interpreted as a single atomic operation, this means that, in our setting, selection
The Bulletin of the EATCS

```
enum Answer = {accepted, denied};

class SaveTheWolf {
    usage lin&{init: lin&{run: un end}};
    Petition p;
    Signatory[Sign] signatory1;
    Signatory[Sign] signatory2;
    unit init(PetitionServer s, 
    Signatory[Sign] s1, 
    Signatory[Sign] s2) {
        p = s.newPetition();
        signatory1 = s1;
        signatory2 = s2;
    }
    unit run() {
        p.setDate(new Date(31, 12, 2010));
        p.setTitle("Save the Wolf");
        p.setDate(new Date(31, 12, 2100));
        switch (p.submit()) {
            case Answer.accepted:
                fork signatory1.signPlease(p);
                fork signatory2.signPlease(p);
                p.sign("me");
            case Answer.denied:
                free p;
        }
    }
}

class Signatory {
    usage lin&{setName: Sign} where
    Sign = un&{signPlease: Sign};
    string name;
    unit setName(string n) {
        name = n;
    }
    unit signPlease {
        (Petition [Promotion] p) {
            p.sign(name);
        }
    }
}

class PetitionServer {
    Petition newPetition() { new Petition(); }
}

class Petition {
    usage Setup where
    Setup = lin&{setTitle: Setup, 
    setDate: Setup, 
    submit: lin&{accepted: Promotion, 
    denied: lin end}}
    Promotion = un&{sign: Promotion, 
    howMany: Promotion};
    string title = "Save me";
    Date date = new Date(1,1,1970);
    List signatures = new List();
    unit setTitle(string t) { title = t; }
    unit setDate(string d) { date = d; }
    Answer submit() { Answer.accepted; }
    sync unit sign(string name) {
        signatures.add(name);
    }
    int howMany() { signatures.length(); }
}

class Main {
    unit main() {
        PetitionServer server = new PetitionServer();
        Signatory s1 = new Signatory();
        s1.setName("signatory1");
        Signatory s2 = new Signatory();
        s2.setName("signatory2");
        SaveTheWolf wolf = new SaveTheWolf();
        wolf.init(server, s1, s2);
        fork wolf.run();
    }
}
```

Figure 8: Petition example in an object-based language
followed by output is also atomic and we can take advantage of this situation to simplify interaction in unrestricted mode (see below). Having mapped the four main operations of session types into OO concepts, it remains to discuss how to enforce protocols running on object references: we use for the effect a usage annotation in classes.

Let us then analyse the code. In order to model the possible outcome of a submit operation we setup an enumeration in line 1, Figure 8. This time we start by describing the server side. Class PetitionServer has one single purpose: to create Petition references, and it does this whenever invoked at method newPetition (line 2). All our classes are equipped with usage annotations, even if inserted by the compiler. In this case, method newPetition is repeatedly available, hence the (implicit) annotation is Init where Init = un&{newPetition: Init}, which we abbreviate to +&{newPetition}, following the schema introduced in Section 2.

Class Petition follows the protocol described by its usage clause, lines 5–11. With respect to type Petition, Section 2, we see that input (?) and output (!) are not explicitly present for they can be read from method signatures. We take the chance to make the protocol a little more realistic, by allowing, in the Promotion phase and in addition to operation sign, an operation howMany to obtain the number of signatures obtained so far. This is made possible by the fact that branch followed by input is a natural atomic operation in object-oriented languages. This is what we meant above by the extra flexibility provided by method calls.

The interesting part of the Petition class is exactly the (unrestricted) Promotion phase, involving methods sign and howMany, where the object may be held by multiple signatories. Being in an unrestricted phase, the object usage is of type +&{sign,howMany}, an abbreviation for the type in lines 10–11. But because the object may be held by multiple threads, it is up to the programmer to control concurrency, if so desired. In our case, we allow concurrency in the read method howMany, but prevent concurrent accesses to the write method sign by prefixing the method name with a sync qualifier.

We start the analysis of the client code by reading class Signatory. Method setName plays the role of a constructor, as witnessed by usage lin&{setName: Sign} in line 31. After initialization signatories can be shared while becoming ready to sign multiple petitions, via method signPlease. The petition to be signed is given as a parameter of type Petition [Promotion], an abbreviation for the type in lines 10–11, class Petition. This means that the reference received as parameter must have been subject to the Setup phase (lines 6–9, class Petition).

Class SaveTheWolf comprises two methods: init and run. They are supposed to be run in sequence; in fact init plays the role of an object constructor, a notion our language is not provided with. We decided that objects of this class must be initialised only once and run only once. This is enforced via the class annotation usage lin&{init: lin&{run: un end}}, where end can now be short for &{}, meaning...
that no further method is available to threads holding a reference to the object.

Rather than receiving the petition medium over a well-known channel (ps in
the previous sections), method init asks the petitionServer for one such reference
(line 10). The method also accepts two previously initialized signatories of type
Signatory[Sign], defined in line 32. Method run conducts petition setup. Notice the
presence of the switch construct to branch accordingly to the result of the submit
operation (lines 20–26). The subsequent usage of reference p crucially depends
on this test: if accepted then the protocol moves to the promotion phase, where the
reference is distributed to two signatories and used to convey the signature of the
petition creator itself (lines 22–24); if denied then reference p is at state lin end
and we use operation free to release the memory allocated by the object, since no
further operation is possible on the reference (witnessed by pre-type end) and p is
the only reference to the object (described by the qualifier lin). Lines 22–23 fork
two threads each running the code of method signPlease in class Signatory.

It is instructive to compare the “end” part of the session types for classes
SaveTheWolf and Petition, unrestricted for the former, linear in the latter. The only
reference to the only object of class SaveTheWolf is wolf in line 9, Main class. The
reference is used to fork a thread (running the code of method run); if we assume
that the fork operation succeeds immediately without waiting for the completion
of method run, then method main cannot free the object, for its code might still
be in use in the thus created thread. The case for reference p of type Petition is
different: in line 26 we know that there is no thread running code of the object and
may thus release the memory.

Finally, class Main creates a petition server, two signatories and forks a thread
to run the SaveTheWolf petition client.

Typing classes. The type checker of our language makes sure that a) client code
calls methods in the order specified in the class usage type; b) client code tests
method results, if applicable, before proceeding to the next call; and c) references
to linear objects are consumed to the end before being freed. Type-checking is
modular and performed following a top-down strategy: program checking is con-
ducted by checking each class separately, which in turn conducts the checking
of each method within the class in the order in which it appears in the classe’s
usage type. The type system keeps track of the state of each field, each param-
eter and each local variable. When we call a method $m$ on a field $o.f$, we check
that the context associates to $o.f$ a type $q & \{ m : T, \ldots \}$, and we update the type
of the field to $T$. When switching on the result of a method call we check that
the reference on which the method was called is of type $q \oplus \{ c_1 : T_1, \ldots, c_n : T_n \}$,
where $c_1, \ldots, c_n$ are the constants in the enumerated type returned by the call. The
context for branch $c_i$ is then updated with type $T_i$. The technical details can be
found in references [5, 1]; a prototype is available online.

Acknowledgements. This work was supported by FCT/MCTES via projects PTDC/EIA–CCO/105359/2008 and CMU–PT/NGN44–2009–12. The author would like to thank Joana Campos, Mariangiola Dezani-Ciancaglini, Simon Gay, and Marco Giunti for insightful comments.

References


The Bulletin of the EATCS


THE DISTRIBUTED COMPUTING COLUMN

BY

PANAGIOTA FATOUROU

Department of Computer Science, University of Crete
P.O. Box 2208 GR-714 09 Heraklion, Crete, Greece
and
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
N. Plastira 100. Vassilika Vouton
GR-700 13 Heraklion, Crete, Greece
faturu@csd.uoc.gr
FAILURE DETECTORS TO SOLVE
ASYNCHRONOUS k-SET AGREEMENT:
A GLIMPSE OF RECENT RESULTS

Michel Raynal
Senior Member, Institut Universitaire de France
IRISA, Université de Rennes, 35042 Rennes Cedex, France
{raynal}@irisa.fr

Abstract

In the k-set agreement problem, each process proposes a value and has to decide a value in such a way that a decided value is a proposed value and at most k different values are decided. This problem can easily be solved in synchronous systems or in asynchronous systems prone to t process crashes when t < k. In contrast, it has been shown that k-set agreement cannot be solved in asynchronous systems when k ≤ t. Hence, since several years, the failure detector-based approach has been investigated to circumvent this impossibility. This approach consists in enriching the underlying asynchronous system with an additional module per process that provides it with information on failures. Hence, without becoming synchronous, the enriched system is no longer fully asynchronous. This paper surveys this approach in both asynchronous shared memory systems and asynchronous message passing systems. It presents and discusses recent results and associated k-set agreement algorithms.

Keywords: Asynchronous system, Eventual leader, Failure detector, Message passing system, Quorum, k-Set agreement, Shared memory system, Wait-freedom.

1 Introduction

The k-set agreement problem This problem, that involves n processes that may fail by crashing, is a coordination problem (sometimes called decision task) introduced by S. Chaudhuri [11]. Her aim was to explore the relation linking the number of process failures and the minimal number of values that the processes are
allowed to decide. This problem is defined as follows [11, 34, 39]. Each process proposes a value and every non-faulty process has to decide a value (termination), in such a way that any decided value is a proposed value (validity) and no more than \( k \) different values are decided (agreement). The problem parameter \( k \) defines the coordination degree: \( k = 1 \) corresponds to its most constrained instance (consensus problem) while \( k = n - 1 \) corresponds to its weakest non-trivial instance (called set agreement problem).

Let \( t \) be an upper bound on the number of processes that may crash in a run, \( 1 \leq t < n \). Hence, \( t \) is a model parameter. If \( t < k \), \( k \)-set agreement can be trivially solved in both synchronous and asynchronous systems: \( k \) predetermined processes broadcast the values they propose and a process decides the first proposed value it receives. Hence, the interesting setting is when \( k \geq t \), i.e., when the number of values that can be decided is smaller or equal to the maximal number of processes that may crash in any run.

Algorithms that solve the \( k \)-set agreement problem in message passing synchronous systems when \( k \geq t \) are presented in [3, 27, 37]. These algorithms are based on sequence of synchronous rounds. It is shown in these books (see also [12]) that \( \lfloor \frac{t}{k} \rfloor + 1 \) rounds are necessary and sufficient to solve \( k \)-set agreement. (It is shown in [37] that this lower bound is still valid in more severe failure models such as the general omission failure model.) For asynchronous systems, the situation is different. When \( t \geq k \), the \( k \)-set agreement problem has no solution [7, 24, 40].

**The failure detector-based approach** A failure detector is a distributed oracle that gives alive processes hints on process failures [9, 35]. Failure detectors have been investigated to solve the \( k \)-set agreement problem since 2000 [30]. (Random oracles to solve the \( k \)-set agreement problem have also been investigated [31].) Lower bounds to solve \( k \)-set agreement in asynchronous message passing systems enriched with limited accuracy failure detectors have been conjectured in [30] and proved in [23]. The question of the weakest failure detector class for the \( k \)-set agreement problem \( (k > 1) \) has been stated first in [38].

Let \( P \) be a problem that is impossible to solve in a pure asynchronous system. A non-trivial failure detector is a failure detector that allows a problem such as \( P \) to be solved. Implementing a non-trivial failure detector requires that the underlying system satisfies appropriate behavioral assumptions. The interested reader will find such assumptions and corresponding algorithms in [36] (Chapter 7) for asynchronous message passing systems and in [18] for asynchronous shared memory systems.
The Bulletin of the EATCS

Content of the paper The paper is on the use of failure detectors that allows $k$-set agreement to be solved in asynchronous systems prone to process crashes. It is made up of 5 sections. Section 2 presents the process model and defines the $k$-set agreement problem. Then the two main sections of the paper follow. Section 3 considers the case where the communication medium is a read/write shared memory. It presents the weakest failure detector for the $k$-set agreement problem in such a setting. This failure detector, denoted $\Omega_k$, which was conjectured to be the weakest in [33], has been proved to be the weakest in [20]. A corresponding $k$-set agreement algorithm is also presented in that section. “Weakest” means here that any failure detector that can be used to solve the $k$-set agreement problem in a crash-prone asynchronous shared memory system provides us with information on failures from which $\Omega_k$ can be built. (More formally, showing that a failure detector is as strong as another one is based on reductions, e.g., [6, 10, 14]).

Section 4 considers then the case where the communication medium is a reliable asynchronous message passing network. Maybe surprisingly, the weakest failure detector for solving $k$-set agreement is different in shared memory systems and message passing systems. Moreover, the corresponding weakest failure detectors are known only for the case $k = 1$ (consensus) and $k = n - 1$ (set agreement). This section presents the most recent results known for the other cases, which leaves open the discovery of the corresponding weakest failure detector. Several failure detector proposals and algorithms are also described. Finally, Section 5 concludes the paper. (To keep the presentation simple, the theorems and algorithms are presented without their proof. The reader will find them in the papers in which they have been introduced.)

To help the reader have a global view, Table 1 summarizes the main failure detector classes presented in this paper (FD, SM and MP stand for failure detec-

<table>
<thead>
<tr>
<th>FD class</th>
<th>Introduced in</th>
<th>Presented in Sec.</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Omega$</td>
<td>[10]</td>
<td>4.3</td>
<td>Weakest for Consensus in SM</td>
</tr>
<tr>
<td>$\Omega_k$</td>
<td>[32]</td>
<td>4.5</td>
<td>Solves $k$-set agreement in SM</td>
</tr>
<tr>
<td>$\Upsilon$</td>
<td>[21]</td>
<td>3.2</td>
<td>Sufficient for $(n - 1)$-set agreement in SM</td>
</tr>
<tr>
<td>$\Omega_{n-1}$</td>
<td>[41]</td>
<td>3.2</td>
<td>Weakest for $(n - 1)$-set agreement in SM</td>
</tr>
<tr>
<td>$\Omega_k$</td>
<td>[33]</td>
<td>3.2</td>
<td>Weakest for $k$-set agreement in SM</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>[14]</td>
<td>4.2</td>
<td>Weakest for Register in MP</td>
</tr>
<tr>
<td>$(\Sigma, \Omega)$</td>
<td>[15]</td>
<td>4.3</td>
<td>Weakest for consensus in MP</td>
</tr>
<tr>
<td>$\Sigma_k$</td>
<td>[5]</td>
<td>4.5</td>
<td>Necessary for $k$-set agreement in MP</td>
</tr>
<tr>
<td>$\Lambda$</td>
<td>[16]</td>
<td>4.3</td>
<td>Weakest for $(n - 1)$-set agreement in MP</td>
</tr>
<tr>
<td>$\Lambda_k$</td>
<td>[4]</td>
<td>4.4</td>
<td>Solves $k$-set agreement in MP</td>
</tr>
<tr>
<td>$\Pi_k$</td>
<td>[5]</td>
<td>4.5</td>
<td>Same power as to $(\Sigma_k, \Omega_k)$</td>
</tr>
</tbody>
</table>

Table 1: Global picture: failure detector classes related to $k$-set agreement
tor, shared memory and message passing, respectively). The reader interested in
the computability power and the robustness of $k$-set agreement-oriented failure
detector classes can consult [29].

2 Process model and $k$-set agreement

Process model The system consists of a set $\Pi = \{p_1, \ldots, p_n\}$ of $n$ asynchronous
processes. The integer $i$ is called the index or identity of process $p_i$.

Each process executes a sequence of atomic steps (each of which may contain
any finite amount of local computation and either a read from or a write to the
shared memory in case of a shared memory system, or a message send or receive
in case of a message passing system). A process executes its code until it possibly
 crashes. After it has crashed a process executes no more step.

A run is a sequence of steps issued by processes such that, according to the
communication model, any value read has been previously written or any message
received has been previously sent. A process that crashes during a run is faulty in
that run, otherwise it is correct. Given a run, $C$ denotes the set of processes that
are correct in that run.

>From a notation point of view, local variables are denoted with lowercase
letters and the process index $i$ is used as a subscript.

External global time For presentation and analysis purposes, we assume that
there is a discrete global clock which ticks every time a process takes a step. This
clock is not accessible by the processes.

Let $\text{var}_i$ be a local variable of process $p_i$. $\text{var}_i^\tau$ denotes the value of $\text{var}_i$ at time $\tau$.

Failure pattern The failure pattern associated with a run, is a function $F(\tau)$
that outputs the set of processes crashed at time $\tau$, $\tau \geq 0$. As processes do not
recover, we have $F(\tau) \subseteq F(\tau + 1)$. $\mathcal{F} = \bigcup_{\tau \geq 0} F(\tau)$ (the set of faulty processes in
the corresponding run). Let us observe that $C = \Pi \setminus \mathcal{F}$.

The model parameter $t$, $1 \leq t < n$, denotes the upper bound on the number of
processes that may crash in a run. When $t = n - 1$, the set of all possible failure
patterns is called the wait-free environment. We say that an algorithm wait-free
solves a problem if any correct process terminates with the right result whatever
the number of faulty processes.

Failure detector A failure detector is a device (oracle) that provides each pro-
cess $p_i$ with read-only local variables containing information of process failures
The Bulletin of the EATCS

[9, 35]. According to the quality of this information and the problem they help solve, several classes of failure detectors have been defined [9, 15].

The $k$-set agreement problem As already indicated, the $k$-set agreement problem has been introduced by S. Chaudhuri [11]. It generalizes the consensus problem (that corresponds to $k = 1$). It is defined as follows. Each process proposes a value and has to decide a value in such a way that the following properties are satisfied:

- **Termination.** Every correct process decides a value.
- **Validity.** A decided value is a proposed value.
- **Agreement.** At most $k$ different values are decided.

A process $p_i$ invokes the operation $X.set\_agreement_k(v_i)$ (where $v_i$ is the value it proposes) to participate in a $k$-set agreement instance denoted $X$. This operation returns to the invoking process $p_i$ the value that $p_i$ decides. The $k$-set agreement problem is a *one-shot problem*, which means that each problem instance is independent of the other instances.

3 $k$-Set agreement in asynchronous shared memory systems

This section presents first the class of the weakest failure detectors for $k$-set agreement in crash-prone shared memory systems. This class is denoted $\Omega_k$. The section describes then an $\Omega_k$-based $k$-set agreement for these systems.

3.1 Communication model

The processes communicate by reading and writing atomic registers [25]. This means that all shared memory accesses appear as if they have been executed one after the other and this total order respects the partial order imposed by their execution. From a notation point of view, shared variables are denoted with uppercase letters.

The corresponding shared memory model (in which at most $t$ processes may fail) is denoted by $ASM_t[\emptyset]$. When the system is enriched with a failure detector $X$, it will be denoted $ASM_t[X]$. $ASM_{n-1}[\emptyset]$ is consequently the asynchronous wait-free shared memory model (wait-free because algorithms designed for this model have to be correct and allow correct processes to terminate despite the occurrence of up to $t = n - 1$ process failures).
Despite the fact that the sentence “an algorithm wait-free solves a problem in a system model in which \( t = n - 1 \)” is a pleonasm (as it contains both “wait-free” and “\( t = n - 1 \)”), we voluntarily use it in the following to insist on wait-free solvability.

3.2 The failure detector class \( \Omega_k \)

A failure detector called \( \text{anti-}\Omega \) (denoted here \( \Omega_{n-1} \)) has been introduced by Zielinsky [41] and shown to be the weakest to solve the \((n-1)\)-set agreement problem. As indicated by Zielinsky, the failure detector class \( \Gamma \), that has been previously proposed in [21], was instrumental in the discovery of \( \Omega_{n-1} \). A failure detector of the class \( \Gamma \) eventually informs the processes that, in the current run, some set of processes cannot be the set of correct processes. It is shown in [21] that \( \Gamma \) is sufficient for solving \((n-1)\)-set agreement.

A generalization of \( \Omega_{n-1} \) denoted \( \Omega_k \) has been introduced in [33] where it is conjectured to be the weakest for shared memory \( k \)-set agreement. This conjecture has been proved by Gafni an Kuznetsov in [20].

Definition  A failure detector of the class \( \Omega_k \) provides each process \( p_i \) with a read-only variable denoted \( mv\_leader_i \) (moving leaders) such that the following properties are satisfied.

- Validity. \( \forall i : \forall \tau : mv\_leaders_\tau^i \) is a set of \( k \) process identities.
- Weak Eventual leadership. \( \exists \tau : \exists \ell \in C : \forall \tau' \geq \tau : \forall i \in C : \ell \in mv\_leaders_\tau'^i \).

The weak eventual leadership property states that there is a time \( \tau \) and a correct process \( p_\ell \) such that, after time \( \tau \), no correct process “suspects \( p_\ell \) to have crashed”. Let us notice that the time \( \tau \) is never revealed to the processes. Moreover, no process explicitly knows the fact that a correct process is included (and will stay forever after) in all sets identified by \( mv\_leaders_\tau \), \( \forall i \in C \).

After a correct process \( p_\ell \) remains forever in all sets \( mv\_leaders_\tau \), no process knows this fact explicitly.

\( \Omega_1 \) is the same as \( \Omega \) (the eventual leader failure detector which has been proved to be the weakest failure detector for solving the consensus problem in asynchronous shared memory systems [10, 26]).

Theorem 1. [Gafni-Kuznetsov 2009] 1 When considering failure detector-enriched systems, the model \( \mathcal{ASM}_{n,n-1}[\Omega_k] \) is the weakest asynchronous shared memory

\footnote{1 This result has also been proved in [17] and [19] using different techniques.}
model in which the k-set agreement problem can be wait-free solved. (Proof in [20].)

In this theorem (and following ones), we use the sentence “When considering failure detector-enriched systems” to insist on the fact that we are interested in systems enriched only with failure detectors. This is because one could imagine other possible types of “enrichment” that would allow k-set agreement to be solved in the corresponding enriched systems.

### 3.3 An $\Omega_k$-based k-set agreement algorithm

**Underlying principle** The principle that underlies the design of the $\Omega_k$-based k-set agreement algorithm that follows [1, 41] is pretty simple: each process $p_i$ participates in $k$ independent parallel consensus instances, $p_i$ proposes the same value to every instance and decides the value returned from the first instance that locally terminates. To that end the algorithm is made up of two parts: an algorithm that gives $\Omega_k$ a vector shape denoted $\text{vector}_\Omega_k$, and a $\text{vector}_\Omega_k$-based algorithm that solves k-set agreement.

**From $\Omega_k$ to vector $\Omega_k$** vector $\Omega_k$ is a vector denoted $\text{Omega}[1..k]$ such that (a) each $\text{Omega}[x]$ returns a process identity each time it is called, and (b) at least one $\text{Omega}[x]$ behaves as $\Omega_1$ (which is the weakest failure detector that allows consensus to be wait-free solved in $\mathcal{A}_{\mathcal{S}_{n-1}}[\emptyset]$). This means that there is at least one $\text{Omega}[x]$ that outputs the same correct process $p_\ell$ at each correct process after some finite time.

```plaintext
Task $T_i$ % the task $T_i$ is executed by $p_i$
repeat forever
    set $i \leftarrow \text{my\_leaders}_i$;
    for each $j \notin \text{set}$ do
        SUSPICIONS$[i][j] \leftarrow \text{SUSPICIONS}[i][j] + 1$ end for
end repeat.

when $\text{Omega}[x]$ is queried by $p_i$:
    for each $j \in [1..n]$ do $\text{total}[j] \leftarrow \Sigma_{1\leq x \leq n}\text{SUSPICIONS}[x][j]$ end for;
    let $p_1, \ldots, p_n =$ permutation on the $n$ proc.
    s.t. $(\text{total}[j_1], j_1) < \cdots < (\text{total}[j_a], j_a)$;
    return $(j_x)$.
```

Figure 1: From $\Omega_k$ to vector $\Omega_k$ (code for $p_i$) [41]
The code of the wait-free algorithm that constructs $\text{vector}_\Omega k$ from $\Omega k$ is described in Figure 1. A shared array $\text{SUSPICIONS}[i][1..n]$ is associated with each process $p_i$. Only $p_i$ can write it, but any process can read it. $\text{SUSPICIONS}[i][j]$ contains the number of times process $p_j$ has been suspected by process $p_i$ ("suspected by $p_i$" means here "not belonging to the output of $\Omega k$ invoked by $p_i$").

When process $p_i$ queries $\Omega k[x]$, it first computes the total number of suspicions of every process $p_j$ ($\text{total}[j]$) and then orders the processes from the less to the more suspected. Process identities are used to obtain a total order (let us remember that lexicographical order $(a, i) < (b, j)$ is defined as $(a < b) \lor ((a = b) \land (i < j))$).

The intuition that underlies this algorithm is the following. Let $p_\ell$ be a correct process that, after some finite time, belongs to the set $\text{mv_leaders}_i$ of every correct process $p_i$. (Due to the definition of $\Omega k$, such a process $p_\ell$ does exist.) Consequently, after some finite time, the quantity $\Sigma_{1 \leq i \leq n} \text{SUSPICIONS}[x][\ell]$ stops increasing and the pair $(\text{total}[\ell], \ell)$ will then be one of the $k$ smallest pairs computed by any process.

```plaintext
when operation set_agreement$_k$(v$_i$) is invoked by $p_i$:
  for each $x \in [1..k]$ do $\text{CONS}[x].\text{propose}(v_i)$ end for;
  let $v$ be the value returned by the first consensus instance $\text{CONS}[x]$ that terminates;
  return($v$).
```

Figure 2: Wait-free $\text{vector}_\Omega k$-based $k$-set agreement (code for $p_i$)

The $\text{vector}_\Omega k$-based $k$-set agreement algorithm is described in Figure 2. As already indicated, it consists of $k$ parallel and independent consensus instances denoted $\text{CONS}[1..k]$. Process $p_i$ proposes $v_i$ to each consensus instance and decides the first value returned by any of these instances. The $\Omega$-based consensus instance $\text{CONS}[x]$ uses $\Omega k[x]$ as its underlying failure detector $\Omega$. It is easy to see that at most $k$ values can be decided, and that a process that does not crash decides a value. This is because at least one $\Omega k[x]$ -not known in advance- behaves as $\Omega$ and $\Omega$ allows consensus to be wait-free solved in asynchronous shared memory systems [22, 26].

4 k-Set agreement in asynchronous message passing systems

This section focuses on failure detectors for the $k$-set agreement problem in crash-prone asynchronous message passing systems. In contrast to shared memory sys-
tems, the weakest class of failure detectors for these systems is not yet known. Hence, this section presents the last results in that direction.

The weakest class of failure detectors for $k$-set agreement suited to crash-prone asynchronous message passing systems is known only for $k = 1$ and $k = n - 1$. Let us remember that 1-set agreement is the consensus problem, i.e., the more constraining (or strongest) agreement problem, while $(n - 1)$-set agreement is the weakest in the sense that the processes have to eliminate a single value from the proposed values (when we assume that each process proposes a distinct value).

### 4.1 Communication model

The processes communicate by sending and receiving messages through channels. Every pair of processes is connected by a bidirectional channel. The channels are failure-free (there is no creation, alteration, duplication or loss of messages) and asynchronous (albeit the time taken by a message to travel from its sender to its destination process is finite, there is no bound on transfer delays). The notation \( \text{broadcast } \text{msg\_type}(m) \) is used to send a message $m$ (the type of which is $\text{msg\_type}$) to all the processes. It is a (non-atomic) shortcut for \( \text{for each } j \in \{1, \ldots, n\} \text{ do send } \text{msg\_type}(m) \text{ to } p_j \text{ end for} \).

**Notation** The previous asynchronous message-passing model in which at most $t$ processes can crash is denoted $\text{AMP}_{n,t}[\emptyset]$. When enriched with a failure detector or an additional assumption $X$, it will be denoted $\text{AMP}_{n,t}[X]$. As an example $\text{AMP}_{n,t}[t < n/2, \omega]$ means that, in any run, at least a majority of processes are correct and processes can access a failure detector of the class $\omega$.

### 4.2 From shared memory to message passing

**The main question** It is shown in [2] that $t < n/2$ is a necessary and sufficient requirement on the value of the model parameter $t$ in order to simulate a shared read/write register on top of a crash-prone asynchronous message passing system.

Hence, a fundamental question is: “Which is the weakest failure detector that allows a register to be built in $\text{AMP}_{n,n-1}[\emptyset]$?” This question has been answered by Delporte, Fauconnier and Guerraoui who have shown that $\Sigma$ is this failure detector [14].

**The failure detector class $\Sigma$** A failure detector of the class $\Sigma$ provides each process $p_i$ with a set $q_{r_i}$ (called quorum) such that the set of variables $q_{r_i}$ satisfies the following properties. After a process $p_i$ has crashed (if it ever does), it is assumed that $q_{r_i}$ remains forever equal to $\{1, \ldots, n\}$.
• Intersection. \( \forall i, j : \forall \tau_i, \tau_j : (qr^{\tau_i}_i \cap qr^{\tau_j}_j \neq \emptyset) \).

• Liveness. \( \exists \tau : \forall \tau' \geq \tau : \forall i \in C : qr^{\tau'}_i \subseteq C \).

The first property states that any pair of values of two quorums, each taken at any time, do intersect. The second property states that the quorum of any correct process eventually contains only correct processes. When we look at a \( \Sigma \)-based algorithm, the first property is used to ensure its safety/consistency while the second one is used to guarantee its progress.

**Theorem 2.** [Delporte-Fauconnier-Guerraoui 2010] When considering failure detector-enriched systems, \( \text{AMP}_{n-1}[\Sigma] \) is the weakest asynchronous message passing system model on top of which a shared read/write register can be wait-free built. (Proof in [6, 14].)

More developments and algorithms building a register in \( \text{ASM}_{n,t}[t < n/2] \) and \( \text{ASM}_{n-1}[\Sigma] \) can be found in [37].

### 4.3 The cases of consensus \((k = 1)\) and set agreement \((k = n-1)\)

**The case of consensus \((k = 1)\) in message passing systems** The failure detector \( \Omega \) has been introduced by Chandra, Hadzilacos and Toueg in [10]. It provides each process \( p_i \) with a read-only local variable \( \text{leader}_i \) that always contains a process identity. Moreover, after some unknown but finite time, the variables \( \text{leader}_i \) of all correct processes \( p_i \) contain the same process identity which is the identity of a correct process. As already said, when considering \( \Omega,k \), \( \Omega^1 \) is \( \Omega \). The fundamental result associated with \( \Omega \) is the following.

**Theorem 3.** [Chandra-Hadzilacos-Toueg 1996] When enriching a system with a failure detector \( \Omega \) is the weakest failure detector the system \( \text{AMP}_{n-1}[t < n/2] \) has to be enriched with in order for consensus to be solved. (Proof in [10].)

This result is extended in [14] where it is shown that the pair of failure detectors \( (\Omega, \Sigma) \) is the weakest to solve consensus in \( \text{ASM}_{n-1}[0] \). The corresponding \( (\Omega, \Sigma) \) failure detector provides two outputs, one for \( \Omega \) the other one for \( \Sigma \). Intuitively, \( \Sigma \) is used to simulate a shared memory while \( \Omega \) is used to allow correct processes to terminate. Several consensus algorithms for both system models \( \text{ASM}_{n,t}[t < n/2] \) and \( \text{ASM}_{n-1}[\Sigma, \Omega] \) are described in [36].
these boolean variables satisfy the following properties. (After a process \( p_i \) has crashed (if it ever crashes) its boolean \( \text{alone}_i \) is assumed to remain forever equal to \( \text{false} \).)

- **Stability.** \( \exists i : \forall \tau : \text{alone}^\tau_i = \text{false} \).
- **Loneliness.** \( (C = \{i\}) \implies (\exists \tau : \forall \tau' \geq \tau : \text{alone}^\tau_\ell = \text{true}) \).

The stability property states that there is at least one process \( p_i \), whose boolean local variable \( \text{alone}_i \), remains forever equal to \( \text{false} \), while the loneliness property states that, if only one process (say \( p_i \)) is correct, its boolean local variable \( \text{alone}_i \) eventually outputs \( \text{true} \) forever. Let us notice that nothing prevents the value of a boolean local variable \( \text{alone}_i \) from changing infinitely often (as long as the corresponding process \( p_i \) is neither the one whose boolean local variable remains always false, nor the only correct process \( p_i \) in the the case where \( n - 1 \) processes crash). The main result associated with \( L \) is the following Theorem.

**Theorem 4.** [Delporte-Fauconnier-Guerraoui-Tielmann 2008] When considering failure detector-enriched systems, \( \mathcal{ASM}_{n-1}[L] \) is the weakest asynchronous message passing system model in which \( (n-1) \)-set agreement can be wait-free solved. (Proof in [16].)

An algorithm that solves the \((n-1)\)-set agreement problem in \( \mathcal{ASM}_{n-1}[L] \) is described in [16], where it is also shown that \( L \) is strictly stronger than \( \Omega_{n-1} \) and strictly weaker than \( \Sigma \). This algorithm has given rise to a more general algorithm for \( k \)-set agreement (described in Figure 3) and can be obtained from it by taking \( k = n - 1 \).

### 4.4 The class of failure detectors \( L_k \) (1 ≤ k ≤ n − 1)

More general failure detectors than the pair \((\Omega, \Sigma)\) (that is optimal for \( k = 1 \)) and \( L \) (that is optimal for \( k = n - 1 \)) have also been proposed (e.g., [5]). Unfortunately, none of them is the weakest for \( 1 \leq k < n - 1 \). This section presents one of them proposed by Biely, Robinson and Schmid [4].

The failure detector class \( L_k \) This class of failure detectors is a simple generalization of \( L \). More specifically it holds that \( L = L_{n-1} \). A failure detector of the class \( L_k \) is called \((n-k)\)-loneliness failure detector. It is formally defined as follows.

- **Stability.** \( \exists \) a set of processes \( K : |K| = n-k : \forall i \in K : \forall \tau : \text{alone}^\tau_i = \text{false} \).
- **Loneliness.** \( |C| \leq n-k \) \( \Rightarrow (\exists \ell \in C : \exists \tau : \forall \tau' \geq \tau : \text{alone}^\tau_\ell = \text{true}) \).
As we can see, this failure detector generalizes $L$. While the loneliness property of $L$ detects the case where only one process remains alive forever, the loneliness property of $L_k$ detects the case where at most $n - k$ processes remain alive forever.

An $L_k$-based $k$-set agreement algorithm The $L_k$-based algorithm described in Figure 3 solves the $k$-set agreement problem [4]. This algorithm is based on a sequence of asynchronous rounds ($r_i$ denotes the current round number of $p_i$). The local variable $est_i$ is $p_i$’s current estimate of its decision value. The execution of the statement $\text{return}(v)$ returns the value $v$ and terminates the invocation the set_agreement($v$).

```plaintext
when operation set_agreement$_k(v_i)$ is invoked by $p_i$:
(01) $est_i \leftarrow v_i; r_i \leftarrow 1;
(02) \text{repeat forever}
(03) \text{for each } j \neq i \text{ do send } est(r_i, est_j) \text{ to } p_j \text{ end for;}
(04) \text{wait until } (n-k) \text{ messages } est(r_i, \_\_\_) \text{ have been received;}
(05) est_i \leftarrow \min(est_i, \text{the } est_j \text{ received at the previous line;})
(06) \text{if } (r_i = k + 1)
(07) \text{then for each } j \neq i \text{ do send } dec(est_i) \text{ to } p_j \text{ end for;}
(08) \text{return}(est_i)
(09) \text{else } r_i \leftarrow r_i + 1
(10) \text{end if}
(11) \text{end repeat.}

when (alone, v dec(v) is received):
(12) \text{if } (\text{dec(v) received) then } est_i \leftarrow v \text{ end if;}
(13) \text{for each } j \neq i \text{ do send } dec(est_i) \text{ to } p_j \text{ end for;}
(14) \text{return}(est_i).
```

Figure 3: An $L_k$-based $k$-set agreement algorithm (code for $p_i$) [4]

In each round, each non-crashed process $p_i$ first broadcasts a message $\text{est}(r_i, est_i)$ (line 03) to inform the other processes of its current estimate $est_i$ and waits until it has received $(n-k)$ estimate messages associated with its current round $r_i$ (line 04). When it has received these estimates, it computes the smallest of them including its own estimate (line 06). Then if $r_i < k + 1$, it proceeds to the next asynchronous round (line 09). If $r_i = k + 1$ it broadcasts $\text{dec}(est_i)$ to inform the
other processes on the value it is about to decide (line 07) and then decides it (line 08).

When considering lines 01-11 only, let us observe that $p_i$ can block forever at line 04 if more than $k$ processes crash. Such a permanent blocking is prevented by the use of $\text{accept}$ messages (that ensures that, as soon as a process decides, all correct processes eventually decide), and the use of the failure detector. Let us also observe that the boolean $\text{alone}_i$ of a correct process $p_i$ becomes true when the number of correct processes is smaller than or equal to $n - k$. In that case, this correct process $p_i$ unblocks the situation.

On the power of $L_k$ It is shown in [4] that, for $n > 2$ and $k \geq 2$, $L_k$ is either weaker than or not comparable to $\Sigma$. As (a) consensus can be solved in both system models $\text{AMP}_{n-1}[L_i]$ and $\text{AMP}_{n-1}[\Omega, \Sigma]$, and (b) $\text{AMP}_{n-1}[\Omega, \Sigma]$ is the weakest failure detector-based model in which consensus can be solved, it follows that $L_1$ is not the weakest failure detector with which $\text{AMP}_{n-1}[\emptyset]$ has to be enriched in order to solve consensus.

It also follows that, while $L_{n-1} = L$ is the weakest failure detector for $(n - 1)$-set agreement [16], $L_k$, $1 \leq k < n - 1$, is not the weakest failure detector for $k$-set agreement. But, as shown in the following theorem, $L_k$ seems to be not too much stronger than what is necessary.

**Theorem 5.** [Biely-Robinson-Schmid 2009] Let $2 \leq k \leq n - 1$. $k$-Set agreement can be wait-free solved in $\text{AMP}_{n-1}[L_k]$ but $(k - 1)$-set agreement cannot wait-free solved in $\text{AMP}_{n-1}[L_k]$. (Proof in [4].)

**4.5 An important step: $\Sigma_k$ is necessary for $k$-set agreement**

Where is the difficulty As far as the agreement property is concerned (at most $k$ values are decided), a main difficulty in the quest for the weakest failure detector that solves $k$-set agreement in a message passing system lies in capturing the shared memory properties needed to solve this problem. (Said, differently, while implementing shared registers in a message passing system is stronger than necessary when one wants to solve $k$-set agreement, it is not yet known how to weaken the register properties in such a way that, once these “weak” registers have been implemented in a message passing system, $k$-set agreement could be solved in such a system).

An effort in that direction is presented in [13] where are investigated the relations between the implementation of a register and $k$-set agreement in asynchronous crash-prone message passing systems. Let an $x$-register be a register that (a) is shared by $x$ processes only and (b) is implemented by processes that communicate by exchanging messages.
Let us remember that $\mathcal{AMP}_{n,n-1}[X]$ is $\mathcal{AMP}_{n,n-1}[^0]$ enriched with objects $X$. It is shown in [13] that, for $n/2 \leq k \leq n - 1$, $k$-set agreement can be solved in $\mathcal{AMP}_{n,n-1}[^2(n-k)]$-register] while a $2(n-k)$-register cannot be built in $\mathcal{AMP}_{n,n-1}[k$-set agreement].

**The failure detector class $\Sigma_k$** This failure detector class, that generalizes $\Sigma$, has been introduced by Bonnet and Raynal [5] as an effort to capture the shared memory properties necessary to solve $k$-set agreement in message passing systems. As for $\Sigma$, each process is provided with a local read-only variable $qr_i$. It is assumed that after a process $p_i$ has crashed (if ever it does), $qr_i$ remains forever equal to $\{1, \ldots, n\}$. These variables satisfy the following properties.

- **Intersection.** Let $id_1, \ldots, id_{k+1}$ denote $k+1$ process ids, and $\tau_1, \ldots, \tau_{k+1}$ denote $k+1$ arbitrary time instants. $\forall id_1, \ldots, id_{k+1}, \tau_1, \ldots, \tau_{k+1} : \exists i, j : 1 \leq i \neq j \leq k + 1 : (qr^i_{id_i} \cap qr^j_{id_j} \neq \emptyset)$.

- **Liveness.** $\exists \tau : \forall \tau^j : \exists i \in C : qr^i_{\tau} \subseteq C$.

The liveness property is the same as for $\Sigma$ while the intersection property generalizes the one of $\Sigma$ (we have $\Sigma_1 = \Sigma$). That property states that any set of $k+1$ quorums is such that any two of its quorums intersect whatever the time instants at which the values of these quorums have been obtained. (It is interesting to notice that this intersection property is the same as the one used to define $k$-coteries [28].) The main property of $\Sigma_k$ is the following theorem.

**Theorem 6.** [Bonnet-Raynal 2009] $\Sigma_k$ is a necessary requirement when one wants to wait-free solve $k$-set agreement (with a failure detector) in $\mathcal{AMP}_{n,n-1}[^0]$. (Proof in [5].)

It is also shown in [5] that $\Sigma_{n-1}$ and $L_{n-1} = \mathcal{L}$ are equivalent. This means that $\mathcal{L}$ can be built in $\mathcal{AMP}_{n,n-1}[\Sigma_{n-1}]$ and $\Sigma_{n-1}$ can be built in $\mathcal{AMP}_{n,n-1}[\mathcal{L}]$. The algorithm that builds $\mathcal{L}$ in $\mathcal{AMP}_{n,n-1}[\Sigma_{n-1}]$ is pretty trivial. At each process $p_i$, the boolean local variable $alone$ is initialized to $false$ and is set forever to $true$ when the quorum $qr_i$ becomes equals to $\{i\}$.

The algorithm that builds $\Sigma_{n-1}$ in $\mathcal{AMP}_{n,n-1}[\mathcal{L}]$ is described in Figure 4. At each process $p_i$, $qr_i$ is initialized to $\{i, j\}$ where $j \neq i$. Then, $p_i$ periodically broadcasts an $alive(i)$ message to indicate that it has not (yet) crashed. When $p_i$’s boolean local variable $alone$, becomes true, $qr_i$ is set to $\{i\}$ and keeps that value forever. When it receives a message $alive(j)$, $p_i$ resets $qr_i$ to $\{i, j\}$ if $qr_i \neq \{i\}$. A proof of correctness of this construction is given in [5].

Consequently, $\Sigma_{n-1}$ provides us with a quorum-based formulation of the weakest failure detector to solve $(n-1)$-set agreement. In contrast, while $\Sigma_1$ can be built in $\mathcal{AMP}_{n,n-1}[\mathcal{L}]$, the converse is not true.
The Bulletin of the EATCS

The class of failure detectors $\Pi_k$ This class of failure detectors has been introduced by Bonnet and Raynal in in [5]. It is $\Sigma_k$ with the additional property.

- Eventual leadership. $\exists \tau : \exists LD = \{\ell_1, \ldots, \ell_k\} : \forall \tau' \geq \tau : \forall i \in C : qr^\tau_i \cap LD \neq \emptyset$.

It is shown in [5] that $\Pi_{n-1} = L_{n-1}$ and $\Pi_1 = (\Sigma, \Omega)$. Hence $\Pi_k$ captures in a single formulation the weakest failure detector to solve $k$-set agreement for $k = 1$ and $k = n - 1$. It seems that (unfortunately) $\Pi_k$ is not the weakest class of failure detectors for other values of $k$.

It is also shown in [5] that the class $\Pi_k$ and the class $(\Sigma_k, \Omega_k)$ are equivalent (any failure detector of one class can be used to build a failure detector of the other class). The failure detector class $\Omega_k$ has been introduced by Neiger [32]. This class (that has inspired the definition of $\Omega_k$) provides each process $p_i$ with a set $leaders^\tau_i$ such that the following properties are satisfied.

- Validity. $\forall i : \forall \tau : leaders^\tau_i$ is a set of $k$ process identities.

- Strong Eventual leadership. $\exists LD = \{\ell_1, \ldots, \ell_k\} : (LD \cap C = \emptyset) \land (\forall \tau' \geq \tau : \forall i \in C : leaders^{\tau'}_i = LD)$.

It is easy to see that $\Omega_k$ is strictly stronger than $\Sigma_k$: any failure detector of the class $\Omega_k$ belongs to the class $\Omega_k$ while the opposite is not true.

4.6 What can be done with $\Sigma_x$

A $\Sigma_x$-based $k$-set agreement algorithm In [8], Bouzid and Travers present an interesting $k$-set algorithm for the system model $\mathcal{AMP}_{n,n-1}[\Sigma_x]$. This algorithm (that combines ideas from [13] and [16]) is described in Figure 5. A process invokes set_agreement$_v(v_i)$ where $v_i$ is the value it proposes.
The processes are statically partitioned into $x + 1$ partitions, $A_1, \ldots, A_{x+1}$ (i.e., $\forall y \neq z : A_y \cap A_z = \emptyset$ and $\bigcup_{y \in \Sigma} A_y = \{1, \ldots, n\}$). Moreover their sizes are such that $\forall y \in [1..x] : |A_y| = \lceil \frac{n}{x+1} \rceil$ and $|A_{x+1}| = \lfloor \frac{n}{x+1} \rfloor + (n \mod (x + 1))$.

Unfortunately the previous mechanism is not sufficient to prevent processes from blocking forever. Such a blocking can occur when all correct processes belong to the very same partition. Line 02 is used to prevent such a definitive blocking. As after some finite time $qr$ contains only correct processes, we eventually have $qr \subseteq A_y$ if all correct processes belongs to $A_y$. Hence, if $qr \subseteq A_y$, $p_i$ is allowed to decide its own proposal (line 04) after having informed the other processes with a $\text{dec}(v)$ message (line 03). The proof (see [8]) shows that, due to the quorum intersection property of $\Sigma$, at most $n \mod (x + 1)$ additional values can be decided, from which follows that no more than $x\lfloor \frac{n}{x+1} \rfloor + (n \mod (x + 1))$
are decided.

Theorem 7. [Bouzid-Travers 2010] The algorithm described in Figure 5 wait-free solves the k-set agreement problem in the system model \( \mathcal{AMP}_{n,n-1}[\Sigma_i] \) for \( k \geq n - \lceil \frac{n}{x+1} \rceil \). Moreover, there is no wait-free k-set agreement algorithm in \( \mathcal{AMP}_{n,n-1}[\Sigma_i] \) when the triple \((n, x, k)\) is such that \( k < n - \lceil \frac{n}{x+1} \rceil \). (Proof in [8].)

Remark As we have seen, \( \Sigma_1 \) is the weakest failure detector \( \mathcal{AMP}_{n,n-1}[\emptyset] \) has to be enriched with in order to wait-free build a shared register. The previous theorem shows that \( \lceil \frac{n}{x} \rceil \)-set agreement can be wait-free solved in \( \mathcal{AMP}_{n,n-1}[\Sigma_1] \). Hence, while neither a register nor \( \lceil \frac{n}{x} \rceil \)-set agreement can be built in \( \mathcal{AMP}_{n,n-1}[\emptyset] \), both can be solved in \( \mathcal{AMP}_{n,n-1}[\Sigma_1] \).

In contrast, when we consider the asynchronous shared memory system model \( \mathcal{ASM}_{n,n-1}[\emptyset] \), shared registers are given for free while the weakest failure detector-based model in which \( \lceil \frac{n}{x} \rceil \)-set agreement can be wait-free solved is \( \mathcal{ASM}_{n,n-1}[\Omega_{\lceil \frac{n}{x} \rceil}] \).

Using both \( \Sigma_i \) and \( \Omega_z \) In [8] a k-set agreement algorithm for the system model \( \mathcal{AMP}_{n,n-1}[\Sigma_i, \Omega_z] \) is presented. This algorithm works for \( k \geq x \times z \). Moreover, it is also shown in this paper that there is no k-set agreement algorithm in \( \mathcal{AMP}_{n,n-1}[\Sigma_i, \Omega_z] \) when \( k < x \times z \) and \( n \geq x \times 2z \).

5 Conclusion

This paper focused on the k-set agreement problem in asynchronous systems prone to process crash failures. In such a context, it has considered two different communication models: the read/write shared memory model and the message passing model.

As k-set agreement cannot be solved in these models, the paper has presented recent results when the failure detector-based approach is used to circumvent the previous impossibility. As we have seen, while the weakest failure detector (i.e., the one that provides processes with “as few information on failures as possible”) is known when communication is by read/write shared memory, this is not the case when communication is by message passing. The paper has presented the most recent results in that direction. It is hoped that this paper not only will help readers to better understand the problem and its difficulty, but will also help them in the quest for the discovery of the weakest failure detector for the message passing case.
Acknowledgments

I want to thank Panagiota Fatourou for comments that helped improve the presentation of the paper. I also thank all the colleagues who have proposed the failure detector classes presented in the paper, designed the algorithms based on these failure detectors and proved the corresponding lower bounds. Without them, this short survey would not exist!

References


The Bulletin of the EATCS


THE LOGIC IN COMPUTER SCIENCE COLUMN

BY

YURI GUREVICH

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
gurevich@microsoft.com

ANATOMY AND EMPIRICAL EVALUATION OF MODERN SAT SOLVERS

Karem A. Sakallah
EECS, University of Michigan
karem@umich.edu

Joao Marques-Silva
CSI/CASL, University College Dublin
jpms@ucd.ie

Abstract

The Boolean Satisfiability (SAT) decision problem can be deservedly declared a success story of computer science. Although SAT was the first problem to be proved NP-complete, the last decade and a half have seen dramatic improvements in the performance of SAT solvers on many practical problem instances. These performance improvements enabled a wide range of real-world applications, several of which have key industrial significance. This article surveys the organization of modern conflict-driven clause learning (CDCL) SAT solvers, focusing on the principal techniques that have contributed to this impressive performance. The article also empirically evaluates these techniques on a comprehensive suite of problem instances taken from a range of representative applications, allowing for a better understanding of their relative contribution.
1 Introduction

SAT solving technology has advanced significantly over the past 15 years. There are currently dozens of SAT solvers that extend the basic DPLL framework, explained in the next section, with a combination of algorithmic enhancements and optimized data structures. The advances have been spurred in part by regular international competitions and a growing archive of SAT instances from a wide range of real-world applications. Modern SAT solvers differ in many aspects, but most of them include the following four features that have been shown, through extensive empirical evidence, to be critical for scalability and performance:

- Conflict-driven clause learning [33, 34],
- Search restarts [26],
- Boolean constraint propagation based on lazy data structures [37], and
- Conflict-based adaptive branching [37]

The immediate aim of this article is to briefly recount the development of these features and to experimentally characterize their contribution in solving a suite of 1000 benchmarks chosen from 12 diverse application areas. A larger goal is to stir the interest of the theoretical computer science community in developing appropriate theoretical models that can explain the remarkable performance of these modern solvers, and also to explain why they still fail in some cases. To be sure, some efforts along these lines have already been made, but much more needs to be done. Further progress in this field will certainly benefit from a combination of theoretical and experimental/algorithmic developments.

Following a review, in Section 2, of the major developments in SAT technology, Section 3 describes the twelve configurations we instrumented in the MiniSat [19] solver for the purpose of isolating the contribution of the aforementioned features of modern SAT solvers. Section 4 describes the benchmark suite we chose for this study and discusses our reasoning for the choice. The results of the study are presented and analyzed in Section 5; we also provide, in an appendix, detailed run time distributions that compare the various solver configurations for each benchmark family. The paper ends with some concluding thoughts in Section 6.

2 A Brief History of SAT Solvers

The first practical algorithm for solving the SAT problem is usually credited to Martin Davis and his collaborators Hilary Putnam, George Logemann, and Don-
ald Loveland. The context was the development of automated procedures for proving theorems in quantification theory and entailed a) translation of a quantified first-order logic formula into a sequence of propositional formulas in conjunctive normal form (CNF), and b) checking the resulting formulas for satisfiability. The first version of the algorithm appeared in 1960. It was authored by Davis and Putnam [15] and consisted of three basic rules:

**Rule I** simplifies the CNF formula by eliminating one-literal clauses.

**Rule II** identifies variables that occur only positively or only negatively in the formula and deletes all clauses in which such variables occur.

**Rule III** eliminates variables that occur in both polarities.

In modern terminology, Rule I is referred to as the unit-clause rule and its repeated application to simplify a formula is known as Boolean Constraint Propagation (BCP) [54], whereas rule II is commonly referred to as the pure-literal rule. The bulk of the algorithmic complexity is in rule III which replaces all clauses that mention a particular variable with the resolvents involving this variable. Quoting from [15],

III. Rule for Eliminating Atomic Formulas. Let the given formula $F$ be put into the form

$$(A \vee p) \& (B \vee \bar{p}) \& R$$

where $A$, $B$, and $R$ are free of $p$.

Then $F$ is inconsistent if and only if $(A \vee B) \& R$ is inconsistent.

Note that $(A \vee B) \& R$ is what results when $p$ is existentially quantified from $F$. Note also that $(A \vee B)$ must be “multiplied out” to put it back into CNF $^2$ and that the resulting CNF clauses represent all $p$-resolvents in $F$. The complete algorithm repeatedly cycles through these three rules terminating when the resulting formula evaluates to 1 (resp. 0) indicating satisfiability (resp. unsatisfiability) of the original formula.


$^2$Note that the form $(A \vee p) \& (B \vee \bar{p}) \& R$ is shorthand for

$$(A_1 \vee p) \& \cdots \& (A_n \vee p) \& (B_1 \vee p) \& \cdots \& (B_n \vee \bar{p}) \& R$$

where each $A_i$ and $B_j$ is a disjunction of literals other that $p$ and $\bar{p}$. Thus, $(A \vee B) = (A_1 \& \cdots \& A_n) \vee (B_1 \& \cdots \& B_n) = (A_1 \vee B_1) \& \cdots \& (A_n \vee B_n)$.

$^3$The $p$-resolvent of $(A_i \vee p)$ and $(B_j \vee \bar{p})$ is the clause $(A_i \vee B_j)$ which is the disjunction of all literals in the two original clauses other that $p$ and $\bar{p}$. Note that $(A_i \vee p) \& (B_j \vee \bar{p}) \rightarrow (A_i \vee B_j)$. 

98
While appearing to solve the problem in a linear number of steps, this algorithm hides the fact that the size of intermediate formulas can grow exponentially, thus severely limiting its scalability. In the second version of the algorithm, published in 1962 [14], Davis, Logemann, and Loveland replaced rule III with a splitting rule that they referred to as rule III*. Quoting from [14],

III*. Splitting Rule. Let the given formula $F$ be put into the form

$$(A \lor p) \land (B \lor \overline{p}) \land R$$

where $A$, $B$, and $R$ are free of $p$. Then $F$ is inconsistent if and only if $A \land R$ and $B \land R$ are both inconsistent.

Thus, rather than eliminating a variable as in rule III, the new variant considers, in sequence, the two subformulas $A \land R$ (resp. $B \land R$) which are obtained by fixing the value of the chosen splitting variable to 0 (resp. 1). If the satisfiability of either subformula cannot be immediately determined, splitting is repeated by fixing additional variables. This process amounts to a depth-first exploration of the space of truth assignments and avoids the memory blow-up inherent in the first version of the algorithm. Completeness is insured by maintaining a stack of the subformulas that are created by splitting and by chronologically backtracking to them whenever later subformulas are found to be unsatisfiable. These two variants have traditionally been referred to as DP and DLL. Recently, though, DPLL has been increasingly used to refer to the DLL version and we follow this usage in the remainder of the article.

The DPLL algorithm established the basic architecture for all subsequent complete SAT solvers and identified the major computational steps to be:

**Branching** which is the mechanism for moving forward in the search space,

**(Unit) Propagation** which is the mechanism for deducing appropriate consequences, also known as implications, of branching choices,

**Backtracking** which is the mechanism for retracting from regions of the search space that do not contain satisfying assignments.

Of course, this description leaves many details unspecified including how to select variables and values for branching and how to detect unit clauses and propagate implications.

Until the mid 1990s, implementations of DPLL followed the original algorithm rather closely, differing primarily in the heuristics used for branching [13, 17, 42, 51, 29, 6, 21, 50]. In particular, these early solvers executed chronological backtracking, and their capacity was limited to SAT instances with variable counts.
in the low hundreds. The first major enhancement to DPLL came in 1996 with the debut of the GRASP solver [33, 34]. Building on ideas from the Artificial Intelligence (AI) and Constraint Satisfaction Problems (CSP) communities, GRASP introduced a novel procedure for clause learning from conflicts, i.e., from partial assignments that cause the formula to evaluate to 0. Concepts such as dependency-directed backtracking, learning from failures, and nogoods appeared in the mid to late 1970s in connection with the development of truth maintenance systems for planning [20], circuit analysis [49] and medical diagnosis [45]. Similar ideas, such as backjumping and conflict-directed backjumping, and learning were developed later for solving constraint satisfaction problems [16, 43]. The key insight in all of these approaches is the inevitability of failure in search algorithms, i.e., the near-certainty (and in case of unsatisfiable instances the absolute certainty) that regardless of how clever the branching heuristics are, a search procedure cannot avoid visiting parts of the search space that contain no solutions. These failures occur because the set of constraints that characterize the search space are generally incomplete in the sense that they do not explicitly capture all possible interactions among the variables. In other words, failures, which we will henceforth refer to as conflicts, signify missing constraints. Thus, by analyzing conflicts we can learn additional constraints that will help avoid similar conflicts later on in the search.

Learning from conflicts takes a particularly simple and elegant form in SAT. A conflict is captured as a partial variable assignment, namely the assignment that causes the CNF formula to evaluate to 0. Negating this assignment yields the desired missing constraint in the form of a conflict-induced clause. Having the same form as the original formula clauses, these learned clauses do not require any special treatment and can be processed similarly to other clauses by the search algorithm.

A significant feature of clause learning in GRASP is the way it ties together branching, propagation, and backtracking to create effective learned clauses. To appreciate this, note that a naive form of learning is to simply negate the entire set of branching decisions that led to the present conflict. For deep conflicts, the resulting learned clause would be unnecessarily long and not particularly useful for eliminating future conflicts. Effective learning requires identifying a small set of assignments that are sufficient to expose the conflict. GRASP identifies such a set by keeping track of the implication sequences during propagation which it implicitly captures as an implication graph. Nodes in this graph represent either decision assignments or implications due to propagation. Each assignment is also labeled with a decision level that indicates when it was created. This level is incremented each time a branching decision is made, and all implied assignments caused by that decision inherit that decision’s level. Directed edges in this graph capture unit propagation and are labeled with the unit clauses responsible for propagation. Various cuts in this graph correspond to independent sets of assignments.
each of which representing a “witness” of the conflict. In particular, GRASP introduced the notion of unique implication points\(^4\) (UIPs) which correspond to the dominators in the implication graph and demonstrated that UIP-based learning yields short effective conflict-induced clauses. This style of clause learning also allowed the algorithm to backtrack non-chronologically to the decision level of the latest assigned variable in the learned clause. GRASP also introduced the Dynamic Largest Individual Sum of literals (DLIS) branching heuristic and experimentally demonstrated its advantage over other heuristics in common use at the time [32]. DLIS maintains counts of literals in unresolved clauses and uses for its next branching decision the literal with the largest count. The enhancements introduced in GRASP, particularly conflict analysis and clause learning, enabled, for the first time, the solution of SAT instances with up to tens of thousands of variables and hundreds of thousands of clauses.

The next milestone in the development of modern SAT solvers was triggered by the observation that the run time distributions of DPLL solvers on satisfiable instances tend to have heavy tails when branching is randomized [26]. This should not be too surprising since a satisfiable SAT instance can be solved very quickly given a perfect branching sequence (i.e., given the solution) but can also take an exponential amount of time with a poor choice of branching decisions. To mitigate against this inherent variability in run times, Gomes et al [26] proposed the use of rapid randomized restarts so that a solver can escape from bad regions of the search space. Restarts are typically triggered after the solver performs a given number of backtracks. Run lengths can also be varied between restarts using a variety of policies such as Luby [1] which is based on the sequence 1, 1, 2, 1, 1, 2, 4, ···. Search restarts were later shown to work effectively with clause learning [5, 37], and to be useful even for unsatisfiable instances. In addition, researchers continue to explore various heuristics that can improve the effectiveness of restarts such as adaptive restarts [8] and problem-specific restart heuristics [46].

Modern SAT solvers took their final shape with two further enhancements that were introduced in the Chaff solver [37]. Both enhancements were motivated by the desire to significantly reduce the computational cost of branching and propagation which dominated the overall run time of conflict-driven solvers such as GRASP. The first enhancement replaced counter-based procedures for identifying unit clauses and performing BCP with a clever algorithm that has come to be known as the two-watched-literals scheme. In this scheme, which generalizes a similar idea in the SATO solver [55], the status of a clause is maintained by “watching” just two of its literals that are not currently assigned to 0 regardless

\(^4\)A notion inspired by unique sensitization points from the field of automatic test pattern generation [23].
of how many other literals it may have. The status is updated only when one of
the watched literals is assigned to 0. In that case, another literal that is not cur-
rently assigned to 0 must be found and watched; the clause becomes unit if no
such literal can be found except the other watched literal. The key idea in this
scheme is to lazily update the status of a clause since we only need to know when
it becomes unit; assignments to literals that are not watched are irrelevant and do
not incur any overhead. Furthermore, assigning 1 to a watched literal, or unas-
signing it while backtracking, requires no maintenance. The performance benefit
of this scheme is most prominent when a CNF instance contains many large orig-
inal clauses or, as is more likely, the SAT solver adds large learned clauses as the
search proceeds.

The second enhancement introduced in Chaff was the Variable State Indepen-
dent Decaying Sum, or VSIDS, branching heuristic [37]. Unlike earlier branching
heuristics, VSIDS was designed to leverage clause learning. Specifically, noting
that difficult instances tend to generate many conflicts, and corresponding conflict-
induced learned clauses, the heuristic leans towards choosing variables that occur
with higher frequency in the most recent conflicts. Quoting from [37], VSIDS
consists of the following five steps:

1. Each variable in each polarity has a counter, initialized to 0.
2. When a clause is added to the database, the counter associated
with each literal in the clause is incremented.
3. The (unassigned) variable and polarity with the highest counter
is chosen at each decision.
4. Ties are broken randomly by default, although this is config-
urable.
5. Periodically, all counters are divided by a constant.

The adaptation of branching choices to conflict-driven learning has the additional
benefit of low overhead since variable count statistics need to be updated only
upon the occurrence of conflicts. This is in marked contrast to other branching
heuristics, such as DLIS, which require much more frequent updating of literal
counts.

The above narrative was not meant to be comprehensive. Other enhancements,
such as learned clause minimization [48], learned clause deletion policies [25],
search restart strategies [8, 46], formula simplification [18], and literal phase sav-
ing [40], etc., have been, and continue to be, suggested. However, it is fair to
say that the four enhancements described in this section, namely a) GRASP-like
conflict-driven clause learning, b) restarts, c) Chaff-like two-watched-literal prop-
gation, and d) Chaff-like VSIDS branching, have had the most impact in improv-
ing the performance and capacity of modern SAT solvers. It is now standard to
refer to solvers that incorporate these features as conflict-driven clause learning, or CDCL, solvers.

3 MiniSat Configurations

MiniSat [19] is one of the best implementations of conflict-driven clause learning. It has a history of winning in SAT competitions, is open-source, and is particularly easy to modify and adapt for various uses. In this study we instrumented MiniSat 2.2.0 to run in the twelve configurations listed in Table 1. The base configuration emulates the DPLL algorithm: it uses the DLIS branching heuristic, implements BCP with counters, and backtracks chronologically. The remaining eleven configurations are characterized by the on/off setting of the following four options:

- **Option CL (Clause Learning):** When this option is chosen, MiniSat performs conflict analysis, clause learning, and non-chronological backtracking; when CL is turned off, MiniSat reverts to DPLL-style chronological backtracking search.

- **Option RST (Restarts):** When this option is chosen, MiniSat applies a Luby restart strategy with a default cutoff of 100 conflicts (i.e., run lengths between restarts follow the sequence 100, 100, 200, 100, 100, 200, 400, \ldots).

- **Option 2WL (Two-Watched-Literals):** When this option is chosen, MiniSat uses the watched-literals scheme for unit propagation; otherwise, it detects unit clauses by maintaining counters of 0 literals in each clause.

- **Option VSIDS (Variable State Independent Decaying Sum):** When this option is chosen, MiniSat applies the VSIDS branching heuristic; otherwise, it reverts to the DLIS heuristic.

Since VSIDS relies on clause learning, it is not clear how to apply it when clause learning is disabled. Specifically, in the absence of clause learning, the application of VSIDS has no effect as all literal counts remain at their initial value (zero) causing branching decisions to become totally random. To retain the spirit of VSIDS in this case, we allow a limited form of clause learning, namely we allow conflict clauses to be created in order to update the literal counts that VSIDS depends on, but immediately delete these clauses to prevent them from helping avoid similar conflicts in the future. For comparison purposes, though, we have implemented the “crippled” version of VSIDS which will be referred to as VSIDS-.

\[^5\text{http://minisat.se/Main.html}\]
Table 1: MiniSat Configurations

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CL</th>
<th>RST</th>
<th>2WL</th>
<th>VSIDS-</th>
<th>VSIDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPLL</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>CL</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N/A</td>
<td>N</td>
</tr>
<tr>
<td>RST</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>2WL</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>VSIDS-</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>VSIDS</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>¬ CL-</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>¬ CL</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>¬ RST</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N/A</td>
<td>Y</td>
</tr>
<tr>
<td>¬ 2WL</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N/A</td>
<td>Y</td>
</tr>
<tr>
<td>¬ VSIDS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N/A</td>
<td>N</td>
</tr>
<tr>
<td>CDCL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N/A</td>
<td>Y</td>
</tr>
</tbody>
</table>

Other than turning the above options on and off, we ran MiniSat with default settings for its other parameters, in particular, opt_ccmin_mode = 2 for deep conflict clause minimization, opt_phase_saving = 2 for full phase saving, and opt_rnd_init_act = false which sets initial literal activity to 0.

4 Benchmarks

To evaluate these twelve configurations of MiniSat we assembled a suite of 1000 CNF instances drawn from twelve diverse application domains. Table 2 lists the benchmark families along with the number of instances selected to represent each family. Columns SAT and UNS indicate, respectively, the number of satisfiable and unsatisfiable instances for each family, whereas column UNK indicates the number of instances whose satisfiability status is unknown.

The choice of these particular problem instances was based on a number of factors including:

- Representation of real-world problem domains where SAT had been successfully applied over the last decade and a half.

---

6The status of each instance was determined by consulting publicly-available data at various benchmark archives. We were unable to determine the status of only 28 instances and tagged them with UNK even though they may be known to be SAT or UNSAT.
Table 2: Benchmark Families

<table>
<thead>
<tr>
<th>Family</th>
<th>Instances</th>
<th>SAT</th>
<th>UNS</th>
<th>UNK</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atpg</td>
<td>100</td>
<td>28</td>
<td>72</td>
<td>0</td>
<td>Circuit testing</td>
</tr>
<tr>
<td>bioinf</td>
<td>30</td>
<td>8</td>
<td>12</td>
<td>10</td>
<td>Bioinformatics</td>
</tr>
<tr>
<td>config</td>
<td>50</td>
<td>15</td>
<td>35</td>
<td>0</td>
<td>Product configuration</td>
</tr>
<tr>
<td>crypto</td>
<td>30</td>
<td>26</td>
<td>3</td>
<td>1</td>
<td>Cryptanalysis</td>
</tr>
<tr>
<td>equiv</td>
<td>30</td>
<td>5</td>
<td>25</td>
<td>0</td>
<td>Equivalence checking</td>
</tr>
<tr>
<td>fpga</td>
<td>50</td>
<td>25</td>
<td>22</td>
<td>3</td>
<td>FPGA routing</td>
</tr>
<tr>
<td>hbmc</td>
<td>250</td>
<td>88</td>
<td>146</td>
<td>16</td>
<td>Hardware bounded model checking</td>
</tr>
<tr>
<td>hverif</td>
<td>200</td>
<td>125</td>
<td>75</td>
<td>0</td>
<td>Hardware verification</td>
</tr>
<tr>
<td>netcfg</td>
<td>10</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>Network configuration</td>
</tr>
<tr>
<td>plan</td>
<td>80</td>
<td>51</td>
<td>24</td>
<td>5</td>
<td>Planning</td>
</tr>
<tr>
<td>sverif</td>
<td>120</td>
<td>57</td>
<td>52</td>
<td>11</td>
<td>Software verification</td>
</tr>
<tr>
<td>termrw</td>
<td>50</td>
<td>26</td>
<td>22</td>
<td>2</td>
<td>Term rewriting</td>
</tr>
</tbody>
</table>

Total: 1000 | 461 | 490 | 49 |

- Representation of benchmark archives that are used to rank solvers in SAT Competitions\(^7\) and SAT Races\(^8\).
- Inclusion of a reasonable number of easy problem instances to enable all solver configurations to finish on at least some instances.
- Weighting the participation of each family (in terms of the number of instances representing it) by the relative success of applying SAT solving technology to that family in the recent past.

The atpg, plan, equiv and fpga families represent SAT applications dating from the early and mid 1990s [30, 44, 38]. Of these, atpg is the one domain where SAT solvers have had the most impact. The config family represents automatic product configuration benchmarks [47], an area where SAT has been applied over the years [27]. Most of the remaining benchmarks were taken from the SAT competitions, again reflecting the relative impact of each practical application. The most successful applications of SAT include hardware bounded model checking (hbmc) [9], hardware verification (hverif) [52, 31], and software verification (sverif) [4]. More recent applications include network configuration problems (netcfg) [39], termination problems in term rewriting systems (termrw) [22], cryptanalysis (crypto) [35], and bioinformatics (bioinf) [10, 12]. Finally, it is

\(^7\)http://www.satcompetition.org/.
\(^8\)http://baldur.iti.uka.de/sat-race-2010/.
worth noting that we are not including in this study randomly-generated benchmarks a) because such benchmarks, especially random 3-SAT instances, have been studied extensively, particularly when the clause-to-variable ratio is approximately 4.25 (the so-called phase-transition region) [36], and b) because real-world benchmarks are rarely random. As we argue later, much work is needed to understand the structure of SAT instances that are derived from actual applications and how such structure affects the behavior of modern SAT solvers.

Figures 1 and 2 provide a variety of statistics that can be useful in differentiating among the benchmark families. Specifically, Figure 1 shows the distribution of the number of variables, number of clauses, and clause-to-variable ratio for the instances in each family. As can be seen, the instances cover a wide range with the smallest instance (50 variables and 159 clauses) coming from the hbmc family and the largest (2,270,930 variables and 8,901,845 clauses) representing the netcfg family. The clause size distribution (number of clauses of a given size) is diagrammed in Figure 2 and shows that while most instances consist of 2 and 3 literals, there is a sizable number of instances that consist of large clauses. Interestingly, the above-mentioned largest netcfg instance has a clause with 865 literals.

5 Evaluation

Before presenting and discussing the results of this study, a couple of caveats are in order. In general, to obtain statistically meaningful run time data requires that a SAT solver be applied to $n > 10$ randomly-ordered (both variables and clauses) versions of each CNF instance. Performing such an extensive evaluation was not feasible, however. Still, we believe that the data we collected provide useful indications of trends and can serve as motivation for further investigations. Secondly, we used a time-out limit of 1000 seconds, which is a little over 15 minutes. A longer time out, say one hour, could potentially, though not likely, affect the conclusions of the study.

The experiments were conducted on a cluster of servers at University College Dublin (UCD) consisting of x86_64 3GHz CPUs with 32GB memory and running the Linux operating system. A total of 12,000 separate runs were performed representing the execution of each of the 12 MiniSat configurations in Table 1 on each of the 1000 CNF instances in our benchmark suite. The results of these experiments are summarized in Figure 3 and Table 3.

The cactus plot in Figure 3 is a helpful way of comparing the performance of all twelve solver configurations. The curve corresponding to each configuration represents its run times for each of the 1000 benchmark instances. However, the
<table>
<thead>
<tr>
<th>Family</th>
<th># Instances</th>
<th>DPLL</th>
<th>CL</th>
<th>RST</th>
<th>2WL</th>
<th>VSIDS-</th>
<th>VSIDS</th>
<th>~ CL-</th>
<th>~ CL</th>
<th>~ RST</th>
<th>~ 2WL</th>
<th>~ VSIDS</th>
<th>CDCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>atpg</td>
<td>100</td>
<td>44</td>
<td>100</td>
<td>38</td>
<td>44</td>
<td>13</td>
<td>82</td>
<td>62</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>94</td>
<td>100</td>
</tr>
<tr>
<td>bioinf</td>
<td>30</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>14</td>
<td>9</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>config</td>
<td>50</td>
<td>49</td>
<td>50</td>
<td>49</td>
<td>49</td>
<td>0</td>
<td>7</td>
<td>2</td>
<td>50</td>
<td>50</td>
<td>29</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>crypto</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>27</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>equiv</td>
<td>30</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>23</td>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td>fpga</td>
<td>50</td>
<td>25</td>
<td>42</td>
<td>24</td>
<td>25</td>
<td>1</td>
<td>25</td>
<td>12</td>
<td>49</td>
<td>45</td>
<td>44</td>
<td>28</td>
<td>47</td>
</tr>
<tr>
<td>hllvm</td>
<td>250</td>
<td>40</td>
<td>187</td>
<td>33</td>
<td>40</td>
<td>18</td>
<td>107</td>
<td>71</td>
<td>127</td>
<td>231</td>
<td>227</td>
<td>164</td>
<td>234</td>
</tr>
<tr>
<td>hverif</td>
<td>200</td>
<td>22</td>
<td>167</td>
<td>21</td>
<td>22</td>
<td>5</td>
<td>119</td>
<td>142</td>
<td>150</td>
<td>197</td>
<td>194</td>
<td>183</td>
<td>198</td>
</tr>
<tr>
<td>netcfg</td>
<td>10</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td>6</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>plan</td>
<td>80</td>
<td>17</td>
<td>44</td>
<td>13</td>
<td>17</td>
<td>7</td>
<td>32</td>
<td>26</td>
<td>48</td>
<td>57</td>
<td>57</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>sverif</td>
<td>120</td>
<td>5</td>
<td>66</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>34</td>
<td>22</td>
<td>73</td>
<td>73</td>
<td>92</td>
<td>82</td>
<td>100</td>
</tr>
<tr>
<td>termrw</td>
<td>50</td>
<td>1</td>
<td>24</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>24</td>
<td>12</td>
<td>22</td>
<td>45</td>
<td>37</td>
<td>34</td>
<td>43</td>
</tr>
<tr>
<td>Totals</td>
<td>1000</td>
<td>204</td>
<td>696</td>
<td>185</td>
<td>204</td>
<td>55</td>
<td>451</td>
<td>364</td>
<td>637</td>
<td>869</td>
<td>847</td>
<td>714</td>
<td>910</td>
</tr>
</tbody>
</table>
data points for each configuration are sorted separately from smallest to largest. In other words, the ordering of the instances along the x-axis is different for different configurations.

Careful study of the data in the table and figure suggests the following conclusions:

1. CDCL, the configuration that includes all four features discussed in this article, provides the best overall performance. It solves 910 out of the 1000 instances within the 1000 second time-out limit.

2. The configurations that represent the absence of a feature suggest that, in terms of effectiveness, the features should be ordered according to CL > VSIDS > 2WL > RST. In other words, fewer instances are solved if CL is missing (637) than if VSIDS is missing (714), etc. Conversely, more instances are solved if that same feature is the only option used by the solver: 696 for CL, 451 for VSIDS, 204 for 2WL, and 185 for RST.

3. The “crippled” VSIDS- option performs extremely poorly, completing on just 55 instances. This is not surprising since, as mentioned earlier, in the absence of clause learning VSIDS degenerates into a random branching heuristic. Option ¬CL– which adds RST and 2WL to VSIDS- improves performance considerably allowing the solution of 364 instances. A possible explanation of this result is that RST is able to overcome the bad branching choices of VSIDS-. This is further corroborated by noting that the average number of restarts under ¬CL– (random branching) is about ten times the average number of restarts under RST (DLIS branching).

4. The RST configuration performs worse than DPLL. This seemed to be anomalous until we realized that the combination of no learning and DLIS branching causes the same decision sequence to be retraced on every restart. Effectively, then, in the RST mode MiniSat is repeatedly executing DPLL until it reaches the conflict cutoff. This also confirms the need to randomize the decision sequence for restarts to be useful. Such randomization occurs naturally when clause learning is enabled since the addition of clauses changes literal activity causing the solver to follow a different decision order on each restart.

5. The performance of 2WL was suspiciously identical to that of DPLL. We expected that 2WL would show some advantage for those instances with large clauses. It turned out, however, that 2WL timed out on those instances and its performance edge on instances with short clauses was not significant enough to outperform counter-based unit propagation. As with search restarts, 2WL is not effective per se, since most problem instances have mostly short clauses.

---

9The data points for instances that timed out are not shown to keep the figure from becoming too cluttered.
However, in the presence of clause learning, the effect of 2WL becomes significant, due to the addition of many large learned clauses.

6. The most significant difference between VSIDS and DLIS is the associated overhead. DLIS requires all literal counters to be updated after unit propagation and after backtracking. Data from the late 1990s indicated that DLIS could account for more than 75% of the run time of GRASP [32]. In contrast, VSIDS has very low overhead. This results in most of the time being spent in the actual search, and so allows solving more problem instances.

7. The most difficult instances come from the bioinf family; only 15 out of the 30 instances were solved. Next are the instances from the equiv family (23 out of 30 solved), the plan family (65 out of 80 solved), and the sverif family (100 out of 120 solved). The instances from the remaining eight families are, comparatively, much easier: all instances from the atpg, config, and netcfg families were solved, nearly all instances from the fpga and hverif families were solved, and over 90% of the instances from the crypto, hhmc, and termrw were solved.

8. An interesting byproduct of this study is that 28 out of the 49 UNK instances were solved. Ironically, 25 of these were solved by the ¬VSIDS configuration: 14 from the hhmc family, 4 each from the plan and sverif families, and 1 each from the crypto, netcfg, and termrw families. The other 3, all from the fpga family were solved by the ¬CL configuration. It is unclear why disabling VSIDS and CL was helpful in these cases.

Additional insights can be gleaned by examining the run time data for each benchmark family separately. Cactus plots for each of the twelve benchmark families are provided in the appendix.

6 Conclusion

The impressive progress in the capacity and performance of SAT solvers over the last fifteen years has accelerated their adoption as indispensable reasoning engines, particularly in hardware and software verification applications. In a recent paper, SAT-based formal methods were reported to have been successfully used as the primary validation vehicle for the Intel Core i7 Processor Execution Engine [28]. In this article we tried to highlight what we think were the primary algorithmic and implementation advances in SAT solving technology that made this possible. But lingering questions remain. With few exceptions, very little has been done theoretically to explain why CDCL solvers work so well. In [7], Beame et al. show that, as a proof system, clause learning is more powerful than regular and DP resolution. Recent work has also shown that a proof system based on clause learning with restarts is as powerful as general resolution (e.g. [41]).
Perhaps an easier question to answer is why CDCL works so well on certain types of problem instances and not so well on others (see observation 8 in Section 5). Studies that characterize the structure of CNF instances can help guide the answers to such questions. For example, intractability may be caused by symmetry and approaches that break symmetry have shown some promise [2]. Other structural attributes such as the cut width of graph representations of CNF instances have been proposed as useful metrics [11]. A recently-suggested promising direction is based on “discovering” that industrial SAT instances have a scale-free graph structure [3] and that this structure seems to be maintained throughout the search process as assignments are made and erased and as learned clauses are added.

In summary, the experimentalists have done quite well in the last few years. What we need going forward is a little bit more theory to explain what has already been done and, perhaps more importantly, to guide the future course of SAT research.

Acknowledgment.

This work was partially supported by SFI grant BEACON (09/PI/I2618) and by the United States National Science Foundation under Grant No. 0705103. The authors wish to also extend their sincere appreciation to Hadi Katebi for his tireless efforts in instrumenting MiniSat and using it to collect and organize the experimental results.

References


The Bulletin of the EATCS


A Detailed Run Time Distributions

Figures 4(a) through 4(l) show, separately, the run time distributions of the twelve MiniSat configurations for each of the benchmark families.
Figure 1: Benchmark Statistics
Figure 2: Clause Size Distribution in Benchmark Families
Figure 3: Run Time Distributions
Figure 4: Detailed Run Time Distributions
Figure 4: Detailed Run Time Distributions (cont.)
Figure 4: Detailed Run Time Distributions (cont.)
Figure 4: Detailed Run Time Distributions (cont.)
THE PUZZLE CORNER

BY

LAURENT ROSAZ

LRI, Orsay CNRS-Université de Paris Sud
Bât 490, 91405 Orsay France
Laurent.Rosaz@lri.fr

Readers are invited to send comments, and to send exercises, even if they don’t know the answer. Write to Laurent.Rosaz@lri.fr.

86  Try to catch the largest

Let \( N \) be an integer. You know what number \( N \) is.

I will pick up a permutation \( \sigma \) of \([1..N]\) at uniform random (each of them may come out with probability \( 1/n! \)). Then I will choose \( N \) numbers \( x_1 < x_2 < \ldots < x_N \). During the next \( N \) minutes, we will do the following: At the beginning of the \( i^{th} \) minute, I will tell you what \( x_{\sigma(i)} \) is. As long as the minute hasn’t ended, you are allowed to claim that the number I just gave to you is the largest one (i.e. \( x_N \)). You are allowed only one guess in the game. If you make a guess and if it is right, you win, otherwise, you lose. Because I choose the numbers, all what you can rely on are the comparisons between numbers. What is your best strategy?

Example. If \( N = 3 \), you may claim that the first number is the largest, which makes you win with probability \( 1/3 \). Or you can do the following: Make no claim in the first minute. If the second number is larger than the first one, then claim the second one is the largest, else if the third one is larger than the first two, then claim the third one is the largest, else you have lost anyway. This latter strategy is better since it makes you win with probability \( 1/2 \) (if the permutation is among \( \{(1,3,2),(2,3,1),(2,1,3)\} \)).
85 What’s next ?

A simple rule leads to the following sequence :

\[ u_0 = 4, u_1 = 3, u_2 = 3, u_3 = 5, u_4 = 4, u_5 = 4, u_6 = 3, u_7 = 5, u_8 = 5, u_9 = 4, \]
\[ ..., u_{100} = 10, ..., u_{999} = 24, u_{1000} = 11, ... \]

What is \( u_{10} \) ?

**Solution** \( u_n \) is the number of letters used to write number \( n \) in english. So \( u_{10} = 3 \).
REPORTS FROM

CONFERENCES
REPORT ON CS&P 2010
Concurrency, Specification and Programming
27–29 September 2010, Landgasthaus Helenenau in Börnicke, Germany
Manfred Kudlek

CS&P 2010, the 19th in this series of workshops, alternating yearly between Germany and Poland, took place about 5 km southeast of Bernau, north of Berlin, in Landgasthaus Helenenau from September 27 to 29, 2010. The lecture rooms had interesting names: Reithof (also dining room), Schafstall, and Rübenkeller.

The organizing committee consisted of Hans-Dieter Burkhard, Gabriele Lindemann, Louchka Popova-Zeugmann, Holger Schlingloff, as well as Nadezhda Govedarova, Heinrich Mellmann, and Jan Wegener. The workshop was attended by 41 participants from 10 countries, details of which given in the following table:

<table>
<thead>
<tr>
<th>Country</th>
<th>BG</th>
<th>CA</th>
<th>CN</th>
<th>DE</th>
<th>IT</th>
<th>PL</th>
<th>RU</th>
<th>SK</th>
<th>SE</th>
<th>UK</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>23</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The scientific program consisted of 40 contributions, 4 of them presented in a plenary session, the others in 2 parallel ones. Details on distribution by countries and number of authors is as follows:

<table>
<thead>
<tr>
<th>Country</th>
<th>BG</th>
<th>CA</th>
<th>CN</th>
<th>DE</th>
<th>IT</th>
<th>PL</th>
<th>RU</th>
<th>SK</th>
<th>SE</th>
<th>UA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>2</td>
<td>23</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The program can be found at http://www2.informatik.hu-berlin.de/ki/CSP2010.

Two contributions could not be presented due to absence of authors, another one was presented by someone else, due to illness of the author. The talks of the plenary sessions were


Andrzej Skowron: Function Approximation: A New Rough Set Approach

Hans-Dieter Burkhard: Narrowing Reality Gap and Validation: Improving the Simulator for Humanoid Soccer Robot
Ludwik Czaja: On Deadlock and Fairness Decision Problems for Computations on Client-server Systems

The workshop covered the fields

<table>
<thead>
<tr>
<th>Mathematical models of concurrency</th>
<th>Parallel algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theory of programming</td>
<td>Object-oriented approaches</td>
</tr>
<tr>
<td>Complexity of parallel algorithms</td>
<td>Knowledge management</td>
</tr>
<tr>
<td>Multiagent systems</td>
<td>Knowledge discovery and data mining</td>
</tr>
<tr>
<td>Algebraical, logical set-theoretic, net-theoretic specification</td>
<td>Soft computing</td>
</tr>
<tr>
<td>Rough sets</td>
<td>Information technology and management</td>
</tr>
<tr>
<td>Specification languages</td>
<td>Applications</td>
</tr>
</tbody>
</table>

The proceedings, containing all contributions, edited by Hans-Dieter Burkhard, Ludwik Czaja, Wojciech Penczek, Gabriela Lindemann, Andrzej Skowron, and Zbigniew Suraj, have been published in two volumes as Informatik-Bericht Nr. 237 of Humboldt-Universität zu Berlin. It is planned to publish a selection of the contributions in a special issue of Fundamenta Informaticae.

The social program consisted of an excursion on Tuesday to the castle and church at Börnicke, where we got coffee and cake, and a dinner at the conference site. Weather was rather chilly, with a lot of rain, and cool. Next CS&P will be held somewhere in Eastern Poland.
European Association for Theoretical Computer Science
EATCS

HISTORY AND ORGANIZATION

EATCS is an international organization founded in 1972. Its aim is to facilitate the exchange of ideas and results among theoretical computer scientists as well as to stimulate cooperation between the theoretical and the practical community in computer science.

Its activities are coordinated by the Council of EATCS, which elects a President, Vice Presidents, and a Treasurer. Policy guidelines are determined by the Council and the General Assembly of EATCS. This assembly is scheduled to take place during the annual International Colloquium on Automata, Languages and Programming (ICALP), the conference of EATCS.

MAJOR ACTIVITIES OF EATCS

- Organization of ICALP;
- Publication of the “Bulletin of the EATCS;”
- Award of research and academic careers prizes, including the “EATCS Award,” the “Gödel Prize” (with SIGACT) and best papers awards at several top conferences;
- Active involvement in publications generally within theoretical computer science.

Other activities of EATCS include the sponsorship or the cooperation in the organization of various more specialized meetings in theoretical computer science. Among such meetings are: ETAPS (The European Joint Conferences on Theory and Practice of Software), STACS (Symposium on Theoretical Aspects of Computer Science), MFCS (Mathematical Foundations of Computer Science), LICS (Logic in Computer Science), ESA (European Symposium on Algorithms), SPAA (Symposium on Parallel Algorithms and Architectures), Workshop on Graph Theoretic Concepts in Computer Science, International Conference on Application and Theory of Petri Nets, International Conference on Database Theory, Workshop on Graph Grammars and their Applications in Computer Science.

Benefits offered by EATCS include:
- Subscription to the “Bulletin of the EATCS;”
- Reduced registration fees at various conferences;
- Reciprocity agreements with other organizations;
- 25% discount when purchasing ICALP proceedings;
- 25% discount in purchasing books from “EATCS Monographs” and “EATCS Texts;”
- Discount (about 70%) per individual annual subscription to “Theoretical Computer Science;”
- Discount (about 70%) per individual annual subscription to “Fundamenta Informaticae;”

(1) THE ICALP CONFERENCE

ICALP is an international conference covering all aspects of theoretical computer science and now customarily taking place during the second or third week of July. Typical topics discussed during recent ICALP conferences are: algorithms, computational complexity, game theory, automata theory, formal language theory, logic, semantics, and theory of programming languages, foundations of networked computation, parallel, distributed, and external memory computing, foundations of logic programming, models of concurrent, distributed and mobile systems, software specification, computational geometry, data types and data structures, models for complex networks, theory of security.
(2) THE BULLETIN OF THE EATCS

Three issues of the Bulletin are published annually, in February, June and October respectively. The Bulletin is a medium for rapid publication and wide distribution of material such as:
- EATCS matters;
- Technical contributions;
- Columns;
- Surveys and tutorials;
- Reports on conferences;
- Information about the current ICALP;
- Reports on computer science departments and institutes;
- Open problems and solutions;
- Abstracts of Ph.D.-Theses;
- Entertainments and pictures related to computer science.

Contributions to any of the above areas are solicited, in electronic form only according to formats, deadlines and submissions procedures illustrated at http://www.eatcs.org/bulletin. Questions and proposals can be addressed to the Editor by email at bulletin@eatcs.org.

(3) OTHER PUBLICATIONS

EATCS has played a major role in establishing what today are some of the most prestigious publication within theoretical computer science.

These include the EATCS Texts and the EATCS Monographs published by Springer-Verlag and launched during ICALP in 1984. The Springer series include monographs covering all areas of theoretical computer science, and aimed at the research community and graduate students, as well as texts intended mostly for the graduate level, where an undergraduate background in computer science is typically assumed.

Updated information about the series can be obtained from the publisher.

The editors of the series are W. Brauer (Munich), J. Hromkovic (Aachen), G. Rozenberg (Leiden), and A. Salomaa (Turku). Potential authors should contact one of the editors.
EATCS members can purchase books from the series with 25% discount. Order should be sent to:
Prof. Dr. G. Rozenberg, LIACS, University of Leiden,
P.O. Box 9512, 2300 RA Leiden, The Netherlands
who acknowledges EATCS membership and forwards the order to Springer-Verlag.

The journal *Theoretical Computer Science*, founded in 1975 on the initiative of EATCS, is published by Elsevier Science Publishers. Its contents are mathematical and abstract in spirit, but it derives its motivation from practical and everyday computation. Its aim is to understand the nature of computation and, as a consequence of this understanding, provide more efficient methodologies.

The Editors-in-Chief of the journal currently are G. Ausiello (Rome), D. Sannella (Edinburgh), G. Rozenberg (Leiden), and M.W. Mislove (Tulane).

**ADDITIONAL EATCS INFORMATION**

For further information please visit [http://www.eatcs.org](http://www.eatcs.org), or contact the President of EATCS:
Prof. Dr. Burkhard Monien, Department of Computer Science
Universität Paderborn, Fürstenalle 11, 33102 Paderborn, Germany
Email: president@eatcs.org

**EATCS MEMBERSHIP**

**DUES**

The dues are €30 for a period of one year. A new membership starts upon registration of the payment. Memberships can always be prolonged for one or more years.

In order to encourage double registration, we are offering a discount for SIGACT members, who can join EATCS for €25 per year. Additional €25 fee is required for ensuring the air mail delivery of the EATCS Bulletin outside Europe.

**HOW TO JOIN EATCS**

You are strongly encouraged to join (or prolong your membership) directly from the EATCS website [www.eatcs.org](http://www.eatcs.org), where you will find an online registration form and the possibility of secure online payment. Alternatively, a subscription form can be downloaded from [www.eatcs.org](http://www.eatcs.org) to be filled and sent together with the annual dues (or a multiple thereof, if membership for multiple years is required) to the **Treasurer** of EATCS:

Prof. Dr. Dirk Janssens, Dept. of Math. and Computer Science, University of Antwerp
Middelheimlaan 1, B-2020 Antwerpen, Belgium
Email: treasurer@eatcs.org, Tel: +32 3 2653904, Fax: +32 3 2653777

The dues can be paid (in order of preference) by VISA or EUROCARD/MASTERCARD credit card, by cheques, or convertible currency cash. Transfers of larger amounts may be made via the following bank account. Please, add €5 per transfer to cover bank charges, and send the necessary information (reason for the payment, name and address) to the treasurer.

**Fortis Bank**, Jules Moretuslei 229, B-2610 Wilrijk, Belgium
Account number: 220–0596350–30–01130
IBAN code: BE 15 2200 5963 5030, SWIFT code: GEBABE BB 18A