News from New Zealand

BY

C. S. Calude

Department of Computer Science, University of Auckland
Auckland, New Zealand
cristian@cs.auckland.ac.nz

1 Scientific and Community News

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

421. A. Raichev. Leinartas's Partial Fraction Decomposition, 06/2012

422. A.A. Abbott, C.S. Calude, J. Conder and K. Svozil. Kochen-Specker Theorem Revisited and Strong Incomputability of Quantum Randomness, 07/2012

423. M.J. Dinneen and Y.-B. Kim. A New Universality Result on P Systems, 07/2012

424. M.J. Dinneen and K. Wei. On the Analysis of a (1+1) Self-Adjusting Memetic Algorithm, 08/2012

2 A Dialogue with David Harel: From theoretical computer science to behavioural programming, biology and smell

Professor David Harel, `http://www.wisdom.weizmann.ac.il/~harel`*, is incumbent of the William Sussman Professorial Chair at the Weizmann Institute of Science and until very recently was the head of the John von Neumann Minerva Center for the Development of Reactive Systems.*

Professor Harel has received honorary degrees from the University of Rennes, the Open University of Israel, the University of Milano-Bicocca and, very recently, from Eindhoven University of Technology. He is a Fellow of the ACM, the IEEE, and the AAAS, and is a member of the Academia Europaea and the Israel Academy of Sciences and Humanities. His long list of awards includes the ACM Karlstrom Outstanding Educator Award (1992), the Stevens Award in Software Development Methods (1996), the Israel Prize (2004), the ACM SIGSOFT Outstanding Research Award (2006), the ACM Software System Award (2007), the ACM SIGSOFT Impact Paper Award (2008), and the Emet Prize (2010).

Cristian Calude: How was computer science during your studies at Bar-Ilan University (BSc), Tel-Aviv University (MSc) and MIT (PhD) in the seventies?

David Harel: At Bar Ilan University in the early 1970's, I took mathematics as a major and computer science as a minor (there was no major in CS then). Studies in CS were mostly technical: languages, programming, operating systems, etc., but there was also a wonderful course on computability and automata given by Yaakov Choueka. Incidentally, Moshe Vardi and Nachum Dershowitz were my contemporaries there, as was Assaf Marron, who now works in my group at Weizmann as a researcher. I remember reaching a point in the first year there (before taking Choueka's course), where I said to myself, "OK, great. I can now write a program that computes the average of a series of numbers; what else is there to learn in computer science?". So a CS degree at that time was very different from what one would get these days, especially for someone with an inclination towards more mathematical topics. Our entire final year, by the way, was jeopardised by the 1973 war, for which I was mobilised as a reserve officer for nine months; we had to take all the senior year courses during the summer of 1974...

At that point, since my impression of CS was that it was more or less just programming and systems, I started an MSc program in algebraic topology at Tel Aviv University. After a year of courses, I was about to give up, when a chance conversation with an elderly numerical analyst, Phillip Rabinowitz, changed my

life: He suggested that I have a chat with Amir Pnueli, who was then a young professor at Tel Aviv University; which I promptly did. I ended up switching to CS on the spot, doing my MSc thesis under Amir's supervision on program proving and the logic of programs. That's when I started to get a deeper appreciation for the theoretical facets of computer science.

MIT was fine, though I kind of rushed in and out, getting the PhD twenty months after arriving. (This was more out of necessity than out of talent: I simply had a young family with two small children and had to start earning fast...) My time there under the supervision of Vaughan Pratt and Albert Meyer was extremely eye-opening. Vaughan and Albert are two very different personalities, both brilliant in their own different ways. This work launched my career. And taking courses from the likes of Ron Rivest and Ed Fredkin didn't do any harm either...

CC: I remember reading and using a nice paper by Choueka and Amir on loop-programs and polynomially computable functions published in 1981. Did Choueka course include this topic?

DH: That's indeed a really nice paper, and I have taught its contents myself. However, without using quantum-driven time machines there is no way Choueka could have included it in our course, which took place in 1972...

CC: The first period of your academic activity was devoted to theoretical computer science: computability theory, logics of programs, database theory, and automata theory. Can you choose one result from each area and comment it?

DH: In Computability, I'd mention two. The first is the *STOC* 1979 (journal version: *JCSS* 1980) completeness result for computable queries, with Ashok Chandra. At the time, we presented this as work on the theory of database queries, but it is actually a lot more general. We like to think of it as a natural extension of the classical Turing-Church-Gödel notion of computability, from words and numbers to general (not-necessarily-ordered) structures. Good examples are graphs or relational databases. The important thing was to provide the "right" definition of the set of all computable functions over structures. We did this by requiring that in addition to having to be computable in the classical sense one requires the functions to preserve isomorphisms, so as not to use information that is not available in the structure itself. We were then able to prove that there a simple complete language (which can be viewed as the first-order relational algebra or calculus augmented with simple loops), which captures exactly the set of computable queries. The second result in computability I'd like to mention is the high undecidability of detecting Hamiltonicity in recursive graphs, from*STOC* 1991. This is a Σ_1^1/Π_1^1 result, and I particularly like it because of its rather "cute" picturesque proof...

In logics of programs, the various axiomatizations of dynamic logic from my 1978 thesis might be noteworthy, as well as the high undecidability of many dif-

ferent logics, such as context free PDL (1981). In automata theory I'd mention the 1988 work (journal version: *JACM* 1994) on extending automata with — in addition to nondeterminism and classical and-parallelism — the new notion of bounded coordinated concurrency, which originates in the language of Statecharts. This comes with a set of results on the exponential power of succinctness that this gives, over and above that offered by nondeterminism and parallelism.

CC: Are there any results you obtained in theoretical computer science that have proved useful in applications?

DH: Not directly, although I like to think of the work with Chandra on computability over structures, and a year later on the complexity theory thereof, as something that may has influenced practical work on query languages for databases. Also, I think that the aforementioned work on the succinctness of Statecharts viewed as automata might have influenced the seriousness with which languages like Statecharts are taken by engineers out there in the real world.

CC: What was your motivation to switch from theoretical to more applied computer science?

DH: Well, I didn't actually switch; there was no conscious decision. In 1983, I was invited to consult for the Israel Aircraft Industries on an avionics project, which is when I started to think more seriously about the problems of specifying the behaviour of what Pnueli and I later identified as *reactive systems*. It was then that I was able to come up with the Statecharts language and the methods around it. (This process is described in a rather personal 2007 *HOPL* paper, with a short version appearing in *Comm. ACM* in 2009.) Over the years, this line of work became more central to my overall activity. The turning point can be said to have occurred with the publication of my last pure theory paper, the 1991 one on Hamiltonicity in recursive graphs. Since then, the only theory I have done is directly related to the practical issues that come up in the other areas of my work.

So, returning to the phrasing of your question, I think that in most cases this kind of move isn't done by premeditated motivation; you simply get caught up in it. Of course, seeing that people actually use your work in real world applications can have a more conscious influence on one's decisions about future research. In addition, it's often an individual issue of talent and ability: I've never thought of myself as being able to do the deepest and most profound work in theoretical computer science. Thus, in time, the more practical facets of our field seemed to be more fitting.

CC: What is the language of Statecharts and, more generally, behavioural programming?

DH: Well, this is very hard to answer in a brief interview like this. Both the language of Statecharts (1984) and the more recent language of *live sequence charts*, LSCs, (1999), the latter defined together with Werner Damm and later extended

with Rami Marelly, are visual formalisms. They are languages for specifying reactive behaviour, which are both formal and rigorous like any other language or any other mathematical concept, but are also inherently visual and diagrammatic. The visuality does not come from using icons, but from using topological and geometric artefacts in the syntax.

We have recently started to use the term *behavioural programming*, as a more general way to capture a programming paradigm that allows people to program complex reactive systems in a way that is driven by, or oriented towards, behaviour, rather than structure. Perhaps I should point the reader to a 2008 *IEEE Computer* paper on “liberating programming" and a recent *Comm. ACM* paper (July 2012) on behavioural programming, in order to get the gist of the idea. In this approach, modularity is not necessarily achieved by the structure but can be done by behaviours. You don’t have to think of your system’s behaviour as being “chopped up" into objects or tasks or components; you can chop it up any way you want according to the way you like to think about the behaviour. These ideas were inspired by the rich structuring of statecharts into multi-level states, but really took off with the live sequence charts. LSCs offer the ability to program using highly modular scenarios, which can be broken up any way you want, but which are also very modal, in the sense that the behaviours can be specified as necessary, possible, forbidden, etc. Something that is not allowed is considered a first-class-citizen behaviour, just like something that has to be done and which you imperatively instruct the program to do. The modularity and the modality are both taken into account in full in the execution engines that run behavioural programs.

In addition to LSCs, we have also non-visual versions of the behavioural programming approach. Of particular interest is a Java version, which allows one to program behaviourally with standard Java tools and compilers, enriched with a small new Java library. Details can be found in the *Comm. ACM* paper.

In this way, our work is a step towards liberating the programmer not only from the need to specify behaviour according to the structure of the program or the system, but also of the need to specify two artefacts — what you want your computer to do and your expectations of the program, i.e., its requirements — since both can actually be part of the same fully executable artefact.

CC: Over the years, your work on languages and methods for software and systems was accompanied by your participation in the design of the tools Statemate (1984-1987), Rhapsody (1997) and the Play-Engine (2003).

DH: Yes, I’ve always made a point of accompanying this kind of research with the building of actual tools. Statemate and Rhapsody were built in a company I co-founded in 1984, I-Logix, which was later acquired by Telelogic and which, as of 2006, is part of IBM. So both of these tools are now IBM products and I’m

no longer involved in any of that. The Play-Engine, and our more recent tool, PlayGo, were built here at the Weizmann Institute by our group. They are not commercial and are available to download, but hopefully one day too they will become (or will inspire) commercial tools. Statemate and Rhapsody essentially are the central tools for using Statecharts in the non-object oriented and object oriented versions, respectively. The Play-Engine and PlayGo are geared towards live sequence charts and behavioural programming.

CC: Please tell us about your "sniffer" and, more generally, about your work on the synthesis and communication of smell.

DH: Actually the work is not about a sniffer per se. The scientifically significant facet of my work on olfaction has to do with trying to solve the major problem in this area, which is the ability to carry out the olfactory analogue of taking a digital photograph and then printing the picture on a piece of paper. Here you would first of all need an input device, a sort of electronic nose, the sniffer, which can take some kind of digital signature of any odour, even an unknown one. Then comes the interesting part: Complicated mathematics and algorithmics have to be carried out, following which a set of instructions is given to an output "printing" device, which we call the whiffer, to emit a particular mixture of a pre-determined set of odorous materials that are inside the whiffer (the "ink" colours). This emitted mixture must have the property of being as close an approximation of the original odour as is possible under the whiffer's circumstances. Now, since odour is not sensed by waves, unlike audio or video, you cannot simply do a Fourier or reverse-Fourier transform. We sniff molecules, and so your system must whiff molecules, otherwise the human nose and brain will not sense them. So the problems are a lot more difficult. They involve psycho-physical issues, comparing human perception with the chemical set up and structure of odorants, and many other things.

Around a decade ago, with two colleagues, I published a viable scheme for how this can be carried out. Still, the scheme requires a tremendous amount of research and understanding about olfaction, much of which is not yet available. On the other hand, we've had some exciting results published on "understanding" odour, including a paper a few years ago showing that it is possible to artificially predict one of the main components of our perception of odour — pleasantness. So by sniffing with an electronic nose, and with a high rate of success, you can tell how pleasant an average human would rate a given odour. Still, it is a long road ahead before we can achieve the grand challenge of being able to communicate and synthesise unknown odours; that is, to sniff them and reproduce them in a different location accurately and reliably.

CC: Few top researchers spend time in academic administration, but you did as head of the department, then dean of your Faculty, and then head of a research centre. How did you manage? What did you achieve?

DH: Well, first of all, in our particular institute these tasks are not as difficult as in, say, a large university. The main reason is that we do not have undergraduates. So one could say that roughly being Dean of the Faculty of Mathematics and Computer Science at the Weizmann Institute is similar to being a department head in a conventional university, because the Faculty is relatively small and we have only graduate students, but no undergraduates.

How did I manage? I think I was an OK Dean, but nothing spectacular. Instead of coming to work in the morning thinking "what can I do for the Faculty today?", I'd come in and think "When will I be able to get through this administrative paperwork and all these meetings and be able to sit down to do the things I really like?" I don't think I have some of the characteristics required of a really good manager. This is also one of the reasons I've always refused to head an entire institution. I've been approached to "run" for President of our own institute, and of several universities, and have always refused, not even giving serious thought to such offers. I don't think I could do a very good job there. Also, I'm still heavily involved in research and in the midst of a demanding research program. Thus, even with managerial talent, which I don't have, I wouldn't feel like spending 90% of my time in management.

CC: The citation for your last Honorary Doctorate has the following statement: "He has put forward the grand challenge of liberating system development from the straightjackets of programming." How will programming evolve in the next decades?

DH: Well, again, I would have loved to talk at length about this, but can't. So, again, the reader is referred to the 2008 "dream" paper in *IEEE Computer*, titled "Can programming be liberated, period". Another paper has just been published in *SoSyM*, jointly with Assaf Marron, titled, "The quest for Runware: on compositional, executable and intuitive models". Both give a relatively detailed — if personal — point of view about how programming will evolve in the next decades. But, very briefly, I think that programming should, and will, evolve to become something much closer to the way we "program" other people, as when we bring up our children, supervise our students or give instructions to our employees or to our broker, real-estate agent or whatever. This is the most general notion of programming: to cause other entities to do what we have in mind for them.

I think this kind of intuitive, informal means should be applicable to programming computers too. Sometimes you'd give an explicit instruction, like "do the dishes now!", and sometimes you'd just show by example how something is to be done. Sometimes you'd give a constraint, like "do whatever you want but be home by 11:00pm", and very often you'd give a forbidden instruction, such as, "never do so or so; I forbid this". The idea is to turn programming into an activity that would be natural and intuitive, using, e.g., natural language and very easy-to-

do interactions. Clearly, the programming tools would have to be embodied with a tremendous amount of understanding, learning, theorem proving, verification, analysis, synthesis, and heuristics, all of which would be active under the table, so to speak, turning this informal, intuitive interaction into an actual executable program. I believe that for many kinds of systems this will be the way things will be done in the future, and we have done quite a bit of work in this direction already.

CC: What is your opinion about the central dogma of digital computing: "computers cannot compute without programmers"?

DH: I don't have a strong opinion about this, as it seems more like a philosophical question. There will always be programmers. The question is whether in the future they will do what most of them do today, which is to write code (or in the best case, to come up with an algorithm and then code it, or to write pieces of code that have to interact with other pieces of code). I think the activity of future programmers will be on a much higher level of abstraction, and will be far more "objective-oriented" (if I may use a pun), with a much larger amount of their time devoted to the deeper issues of how you want your system to behave, leaving a lot of the details about how that is to be achieved to powerful computing tools behind the scenes.

CC: I had pleasure of reading your book *Computers Ltd.: What They Really Can't Do*. Your book *Algorithmics: The Spirit of Computing* as well as the series of lectures on radio and programs on television, have all aimed at a broader audience. What was the response you got?

DH: Thanks for the compliment... I have actually received incredibly good response to this expository work, which includes those two books and some other things too. However, one has to be careful because there is a saying that the worst thing that can happen to a scientist is that he or she is considered a good expositor or populariser... I hope that's not something that people think of seriously when they see this work.

Maybe I should add that what I am most proud of is not a particular book or a particular series of lectures, but the fact that I had the lucky opportunity, relatively early in the computer science game, to decide what in my opinion were the most fundamental issues in computer science. The self-test here was to try to choose those topics that in twenty years would still be relevant, To a large extent I feel that those decisions turned out to be valid. For example, the Algorithmics book was published in 1987 —that is, exactly 25 years ago — and there is now a new printing of a 3rd edition in which very little had to be changed. Of course, there are many new results and new ideas, but apart from programming languages, which have changed quite a bit since the 1980's, and new chapters on system development, most of the issues that were included in the original version are still included in today's edition of that book, almost unchanged. So the test of time

regarding what is fundamental and important seems to have met with success, and that is really my greatest joy in this expository work. People who write about computers and the internet, or about languages and methods, know that 10 years later they have to re-write everything because things will have changed. On the other hand, if you concentrate on algorithms and their complexity, on intractability, undecidability, correctness, and efficiency, and to some extent even on AI, you have to add and extend, but the basic issues remain the same.

CC: What advice would you give a talented student wishing to start graduate studies in computer science?

DH: Hmmm... One thing that comes to mind is not to be afraid of rejections. My 1984 statecharts paper took three years to get published, after being repeatedly rejected from several of the most widely read CS journals, including *Comm. ACM* and *IEEE Computer* (and it now has almost 7000 citations...). Do serious research, have confidence in what you do, and be persistent in exposing it.

Another point is more relevant to undergraduates or to graduate students who have not yet settled on the exact field of research in which they want to work. My advice would be to go for mathematics and computer science, or physics and computer science, and in this day and time maybe the best is to go for biology and computer science. A person with a good background in math and physics, or math and biology weaved with computer science, is prepared for doing cutting edge 21st century science. In fact, perhaps it is best to end this interview by the following statement, which appears in a couple of my recent papers on modelling biology, and which captures my belief about the role computer science will play in the coming years: "Computer science is poised to play a role in the science of the 21st century, which will be dominated by the life sciences, similar to the role played by mathematics in the science of the 20th century, much of which was dominated by the physical sciences".

CC: Many thanks.