

Transcription of the Presentation Is Informatics a Science?

Robin Milner
University of Cambridge, UK

Good afternoon. It's a real honour for me to be part of this celebration. I'm delighted to discover, which I only discovered today, that my scientific life is exactly the same length as INRIA. So I feel very happy about that. I also remember that 35 years ago, when 5 years had already done past for INRIA's life, I met the famous people in the famous building Bâtiment Huit: Maurice Nivat, Jean Vuillemin, Gérard Berry, Jean-Jacques Lévy, Gérard Huet, and of course Gilles Kahn. And I met them in the United States, in Stanford. Since then, I think it's true to say that apart from my own organisations in Edinburgh and Cambridge, I've interacted more closely and more constructively with people at INRIA than at any other time.

I also want to acknowledge the Prefecture of the Ile-de-France for their award of a Blaise Pascal international research chair. The main delight for this was that I spent one year at Écoles Polytechnique in Saclay, with the kind hosts Jean-Pierre Jouannaud and Catuscia Palamidessi, where I was able to spend most of my time trying not to play with words when we ask "Is Informatics a Science?"

It is important that we don't play with words when we're playing with concepts here. We're asking for the importance, or the perhaps lack of importance, of Informatics as a domain of knowledge, as a domain of understanding. Whether that counts as science will be up to you.

We like other scientists, must build models. They are models of what we build. Then we hope to understand what we build, as well as other scientists understand nature. Now the catch is: but of course we don't actually understand some of our existing software systems. I think that must change. But how do we change it?

I'm going to divide my talk into four short pieces. One is why is there such a gap now in Informatics between science and engineering? The second is, in particular for the topic of ubiquitous computing, how can we mend this, or why must we mend this? Then, how we build what I call a "tower of informatic models", because by the time I've finished, I hope that if you aren't already persuaded, you will be persuaded that most of our work, in terms of understanding, is in building models. Then, the fourth part: how do we go about building this tower, because it certainly hasn't been built properly yet.

What about the gulf between Informatics science and engineering? To begin with, software science, or a theory of computation, has developed a number of things. I needn't read them all out. They're things most of us know something about. There's a large conceptual framework within which we can work. Some people work in it without moving out of it, others work in it and do relate it to software. There becomes the evidence of the gulf between engineering and science.

I want to begin by asking: why do we have this gulf? Why is the impact of these concepts on practice so low and haphazard. I found this quotation which is more or less the conclusion

that I've come to and I think many people have: "The pace of technological change, and the ferociously competitive nature of the industry lead to the triumph of speed over thoughtfulness, of the maverick shortcut over discipline, and the focus on the short term." Now this wasn't an Informatics scientist speaking – this was somebody in a totally different field who had simply observed that this was happening. This was quoted in a report which I will say more about shortly, on the challenge of complex IT projects, written in 2005, perhaps in 2004.

Take as an example, the year 2000 problem, or the year 2000 fiasco. There was no disaster, but enormous sums were wasted expecting a disaster, or preparing for a disaster. And yet, as I know well, because I had a small part to play in it, the appropriate theory which could have prevented that in software design and construction had been around for 2 decades. Things often work through from a scientific basis into practice in 2 decades, but not in this case.

What then are the causes of this kind of gulf? These causes I think are things which you can not attribute to any person or group of people. It's something to do with the way our subject has evolved, and the enormous success of the spread of information technology in the world. There are three separate causes at least, I think. Or they are perhaps symptoms of the gulf. First the science is neither complete, nor unified. Well why not? Because the technologies keep changing, and creating more things for us to answer. Second, software houses design for what the markets demand, not what has been subjected to available theoretical analysis. Again, that's under the demand of the market. There's no real other response that they could have made to the market demand without losing the chance to sell product. Third, perhaps as a consequence of all this the software industry focuses on the management of software production without necessarily understanding the concepts of software itself. There is perhaps too much attention paid, of course necessary attention, but in balance as between the management of software production and the very nature of the material of software itself. These are either symptoms or causes, however you want to think, of the gap that we have.

Software engineering is often found faulty. It doesn't match procedures in other engineering disciplines, which tend to be founded on science which has evolved quite slowly. Take the electronics discipline. It goes back at least to James Clark Maxwell, perhaps 100 years, perhaps more. The science has a chance to evolve, and as it were to nurture the technology rather than to arise from the technology.

I was shocked to some extent by a recent report which came out in our country written jointly by the Royal Academy of Engineering and the British Computer Society, which did a wonderful job of recommendations on the software production process, explaining why complex IT projects occasionally do not work as they should. It exposed the fault in great depth, and made very sound recommendations, but hardly mentions the science of software, or the conceptual framework of software at all. The implication there is that software science and engineering are, and possibly should be, disconnected. This disappointed me greatly. I think it's disappointed other people. In fact the authorship of that report did not include some theoretical software scientists. In fact most people on the authorship were not researchers or academic scientists.

It's a grand challenge to establish modelling as the basis of Informatics. Let's see what that might mean. Because in some sense, why do we have to model what we have already built, and therefore presumably what we must understand already?

If you look at the phenomenon of ubiquitous computing, which is partly predicted, partly becoming real, then the challenge there is quite great. Two visions of ubiquitous computing, one from the originator, or shall we say the first and most prominent apostle or predictor or prophet I should say, of the idea. Populations of computing entities will be a significant part of our environment, performing tasks that support us, and we shall be largely unaware of them. It's that last phrase that we need to pay attention to. Supporting that, Josef Weizenbaum, in 2001, is reported as saying (I don't think I have the words exact) "In the next 5 or 10 years the computer will be erased from our consciousness. We simply will not talk about it any longer. We will not read about it, apart from the experts of course". That last phrase again is I think worth our notice. Who are these experts, exactly?

The third vision: "Yes, ubiquitous computing will empower us, if we understand it." This is my qualified vision. It will not empower us if we don't understand it. In fact we possibly run greater risk with ubiquitous computing systems than we ever ran with large software systems of the present day, if we fail to understand.

What are the qualities of a ubiquitous computing system? The notion "ubiquitous" is also synonymous in this situation with the notion of "pervasive". Consider a pervasive computing system; one that pervades our lives. It will continually make decisions which have hitherto been made by us. For example, even maintaining the contents of your refrigerator. Something so banal as that, it will probably do for you. Secondly it will be, in terms of software, very large – orders of magnitude larger than today's systems, many of which of course are in danger of being misunderstood or have become misunderstood because they've been altered frequently. It must adapt continually online to new requirements which are somehow respected, or detected or predicted by it, and perhaps requested by us, or a mixture of the two. For example, if you take a traffic system dealing with or managing driverless cars, and interaction between computers in the cars and on the roadside that may interact with another one which is managing the health in a city and has to get the doctor to an urgent case along a motorway. The ubiquitous computing systems will interact with one another. So those are big differences.

The question is, can the industry cope with traditional software engineering methods? Instead of answering that directly, consider what sort of concepts have to be present in our minds when we're understanding a ubiquitous computing system. Here are just a few of them. In among those you can find one or two which we already are doing things about. Like security, we know about compilation, we know about simulation. How do we work the notions of belief, of intention into complex systems? These are words are used by the community of people working on the intelligent agent systems for example. What about self-management? What about authorisation? A lot of words which go along with security. What about why should one software agent trust another one that has arrived on its doorstep asking for resource? Maybe time, maybe space on something, and so on. I'm sure you can think of more. This is the kind of, against this rich array of ideas, we're going to have to understand these systems working for us.

What do we do in building models of the systems that are going to exist? We have to start by picking out just a few, and I've picked four there, which are rather easy compared with the other ones. That bits of software will be local. For example they'll be local in your body, or on the roadside, or in the car. They will be mobile. Same places again. They will be linked to each other, possibly wirelessly, independently of their location. And there will be uncertainty,

which perhaps will be managed by stochastic interpretation of their behaviour, or stochastic specification of their behaviour.

By picking a few of these concepts, we should begin building models, say a model M built on these four ideas which can then explain one part, an important part of course, of the behaviour of one of these systems. It's a manageable subset of the concepts. So, we would define a model M based on a small manageable subset, for example of those four concepts. But that would be just one of the kinds of model that we build, and then we expect to build models incorporating the other ideas, and as we build those others, they will be likely to be erected in some kind of hierarchical way or possibly be put together side by side so that we'll end with, or proceed to build a tower of models. If you look at what we already do, then you can find a lot of it is already there, in what we already do.

But what do we mean by "model", and what do we mean by "implement", and what do we mean by "tower"? Can we be more specific about this, so that we can first of all detect those things in what we already do, and then perhaps use it as a means of organising software science? If we can do this, then I think we qualify as a science, because we've stratified the understanding of the entities, which happen to be artefacts, artificial entities, that we're trying to understand.

Now, I want to get onto the third section, which is some illustrations in what we already do, and sometimes quite recently have done, in terms of structures of models. The tower of informatic models. What is it? They're going to be inter-related to each other. We've always built models of course. We've built programming languages – that's a model. It's a model that in some sense explains what the machine is doing now. Take Fortran and the informal description of Fortran. That is a model which in some sense gets implemented by the machine design, which is itself a model, which gets built in hardware. Of course, Petri nets are a model. They explain concurrent behaviour in some exquisite detail. Security discipline is a model for behaviour which we either impose or advise to be followed. Intelligent agents are a model which would get implemented by a programming language which might then in turn be implemented on the computer. So you're beginning to see the levelling. You can say more about that. I've got more examples and whatever I've got time for. I'll put some possibly provocative examples up, to indicate how much we already do.

As I'm a theoretician, I've made it my job to talk a little bit, not enough but it's a beginning, with software engineers who are beginning to structure their work in terms of models. Many people here will know at least the phrase model-driven engineering or model-driven architecture. The community of people who design software according to those precepts, are beginning to use the notion of model in a way which I think is consistent with the kind of scientific model which I'm hoping to propose. I believe that there is some consensus or some compatibility between what our engineers and our scientists want to do, or already beginning to do.

What is a model? This is where, of course one has to be provocative. I don't think the answer is easy. It's not accurate. But it has to have some characterisation, so we can say "these are models", "no this is not a model". So what is a model? An example. It was Wittgenstein who said "I can't define a chair, but I know a chair when I see one." One model which we I think all believe in is the notion of a flow chart. But it's not just the flow chart – it's how to execute the flow chart. A model is not just a set of structures, like a flow chart it's how they behave, or how you are to move around in them, or... What is their semantics? A model consists of

some entities and their meanings. And it's this notion of "and their meaning" which is so crucial because without the meaning then why not attach whatever meaning you like to the entities, for example, to programs?

Given that tentative notion of model, but I'm prepared to try and defend it as much as you ask me to, what is a tower of models? The idea is this. That if you have a model say, the Algol60 language, and one of the most beautiful pieces of text that was ever written, the informal description of how it works, you have that model. The question is how do you relate it to other models? You implement it, just like Fortran, and it's implemented on the machine model. There's some notion, there's going to be some notion, there in fact is some notion of validating that implementation whether it's an interpreter, or whether it's a compiler, or whatever it is, we know what it means for it to be doing it properly.

So, greatly daring, I suggest that we use the words "A *explains* B" if A is a model and B is another model, if A is somehow higher, more abstract, than B. If B already exists, then we might say "A *abstracts* B" – think of abstract interpretation. Or if A exists and B doesn't, then it may be said to "*specify*" B. So abstraction and specification are one way of talking about explanation. But it also seems to me that implementation "A *has specified the implementation of* B" or indeed, if A is very vague, and B is a slightly less vague specification, has more detail, then we might say that B, that is the lower model, "*refines*" the other one. We begin to get these different verbs that are used, which we already use. I'm suggesting that we put them all together so that we can build models and call all of the connections "*explanations*". By the way, a logic, together with what it is to infer one thing from another, or what it is for one sentence to be true or false, that's a set of entities and their meaning, and they act as specifications for programs, or explanations for programs.

I've been talking about vertical creation of the tower, but sometimes we combine models side by side. We have models which are informatic models but sometimes in real life systems we have to combine those with mechanical or electronic models of the hardware or the entities that are going to be controlled by the software. We built models together, horizontally. Another example is coordination languages which put programming languages together. They can do it well or badly, and the way that's going to be analysed is in terms of the meanings of the two separately, and the extra meaning that comes from putting them side by side. The tower of models you expect it to be vertical and horizontal. For example, if you combine distributed programs with networks, which have their own many different kinds of models, which are actually some of them far from the notion of programming then you're combining two very different kinds of models into a whole.

Here is a trivial little tower of models. I don't want to spend long on this because it doesn't make much sense. The vertical parts there mean that M is a combination of M1, M2, and M3. M1 *explains* M. M3 in some strange way *explains* itself. But it's not so strange, because M3 may be for example, specifications like in Hoare's language CSP, where one specification *refines* another. In that sense, within one model, its entities *explain* or *refine* each other. That's an abstract example of a tower, but let's go on to something more concrete. The computers themselves, are *realisations* of hardware design, which is the *implementation* of assembly code, which is the *compilation* of programs in another language with its meaning which may have been *specified* by predicate logic.

That, entirely within our experience, is a vertical piece of the tower. I don't want to bore you with detail here, but each pair, that is the four pairs vertically here, this *realisation*, or

implementation, or *interpretation* is as it were an *explanation* and it can be validated. Here are the entities in each case. We needn't read them all. Take assembly code. Its entities are lists of instructions, and its meaning is the way those instructions act on the memory and on the state of the machine, and so on. We needn't look at the rest of those – we sort of know what those entities are without having to think about it very long. But take them pairwise.

Take the bottom pair. How do you test that a hardware design is implemented by a physical machine? Well this is almost like observing the real world. You are observing that the real machine behaves according to your model of it and that's done by observation. I don't think that can be formalised, just as it can't be formalised to confirm or refute the model of quarks or whatever. It can be done incrementally by observation and that's what much science is about.

If you take these things pair-wise, we can go up the list and I won't go all the way. Except that this is quite a nice one, the assembly code, which is a list of instructions, can of course be implemented in gate diagrams in hardware design. Machine-checked proofs, in some of our known systems have been done, which prove the correctness of the computer, or rather of its hardware design. That is, they say that the action on memory is actually correctly implemented by the behaviour in the hardware.

Let's go right to the top here. This is again familiar, but you may not have thought of it like this. If you take a high-level programming language and its meaning is the functions it computes, the types and so on associated with the programs, then logical formulae are used to specify the programs as pre and post conditions. What's a logical formula? What's the meaning of it? It's valuation is something that is true or false, depending on what the free variables, or the names in the formula mean. You've got a model which is predicate logic, or a model which is a programming language, and it turns out that Hoare's well known Hoare logic of triples is the way which you validate that explanation. That's fine. It's not hard to put those things into the tower. That is a tall tower for programming.

Now let's look at something which is very relevant for today, because it's embedded in the history of INRIA and recent history of INRIA. Let's start at the middle of the diagram, and notice that an aircraft design is, or the model for aircraft designs, is a combination of a model for embedded programs, a model for the electro-mechanical design of the aircraft and a model of the environment. A meteorological model, for example. You can't actually explain the aircraft design which is supposed to be how it behaves in the real world, unless you put those things all together. So that's a combination. Then the aircraft designs are of course realised by aircraft themselves each of those separate components – the electro-mechanical design and the programs – are realised.

But up at the top here, is something that's perhaps more interesting for us today, which is the way in which so-called abstract interpretation can be used to understand the embedded programs, the real-time software that it making all the bits of the aircraft work. Let's comment on this a little. The combined model, that's the triple there, environment, electro-mechanical design, embedded programs, can give scientific answers to questions like "At what level of turbulence will the aircraft maintain control?" That's the kind of analysis that is going to be done on that joint model.

After the failure of the launch of Ariane-5, INRIA was, as everyone knows here well, committed by the scientific director who was Gilles Kahn at the time, to analyse the

embedded software for the Airbus. Of course this came after their analysis of the failure of the Ariane-5, and people in INRIA, notably Patrick Cousot and his team, have actually explained the embedded programs, or specified them, using abstract interpretation. This is a wonderful example of model building, because if you look in more detail at that, and this is the key to the notion of abstract interpretation, you don't explain the whole of the behaviour of a program by one higher model or specification, you explain different parts of its behaviour, like "Does it do overflow?", or space overflow, or integer overflow, or real number overflow. You use one abstract model to explain why it behaves correctly in that way, if it does, you use another one for other aspects, and so on. You choose the form of the abstract interpretation in order to explain a particular facet of the behaviour of the programs.

This is, I called it a wonderful example, because it's an example of designing the model in order to explain particular features, rather than taking a model off the shelf, and seeing if it will do the job. It may do of course, in some cases. The great power of this is that explanation is something that is itself a skill, it's I would say a scientific skill, and it's something that's only going to come from deep knowledge both of the programs themselves, and of the possibilities of abstraction that will help you to understand it better. Those two examples give you a sense in which you could make the towers bigger.

What I hope people are not expecting me to do is to say formally what I mean by an explanation, because I think if I were to claim that, then at once the whole idea would lose the credibility that I hope it has. I think the notion of explanation and its looseness, is something that's worth considering, and finding more examples about. But I don't want to attempt a formal definition. This is the kind of thing that leads to a lack of trust between engineer and theoretician. The theoretician can't expect to explain everything, and has to pay attention to the engineering challenge.

That's the end of my detailed examples. Just a remark here that each analysis adopts a particular abstract interpretation, and that's very interesting. What is also interesting, and I'm glad that I remember to say this, is that the task of explanation is something that it itself automated. We all know the successes of model checking, the successes of model checking for abstract interpretation are also remarkable, because they gain by being able to use a different kind of model for different aspects, and they gain efficiency that way. Model checkers, or automated explanations, and there are other examples: automatic proof of compiler correctness, automatic checking of logic specification against the program and so on. Those things are available as tools. We're not going to be able to sell the idea of models to the whole computing community unless we automate that process. Perhaps I shouldn't say "automate". Perhaps I should say provide mechanical tools for assisting the human in making these explanations.

What about building the tower? I've already said most of what I wanted to say about building the tower, but there is time to just mention one or two more examples. The essential thing is to use these models, and their explanations between them, to present our existing knowledge, and also ongoing work. I've got 18 examples here, and I'm just going to flash them up. You've seen some of them already: down the bottom, abstract interpretation abstracts from a programming language, and we've seen all of those already, so I'm going to flip straight on here.

Security is interesting. Security disciplines tend to be realised by cryptography. Security discipline is, well you invent a new key that nobody has ever used before, in cryptography

you worry about is that key going to be decodeable, or how you make sure it's new, and that kind of thing. There's a case where security discipline is realised by or implemented by something lower, but equally important in the hierarchy. Security disciplines themselves are explained in various ways, one of which is mathematical logic. The second line there: higher-order logic explains security disciplines, that's understood.

I'd like to go down to the one nearest to the bottom. Swimlanes, those are otherwise known as message sequence charts. Those are things that even an executive can understand, if you don't mind me saying it in that rather absurd way. In other words, for people who don't have time to understand the technical details of our subject, they can understand message sequence charts. Message sequence charts can explain the intra-communication between members of a community of concurrent processes which perhaps could be implemented in a programming language. Take a programming language which allows many separate components to interact with each other. Such a language is already being designed by the choreography working group within the world wide web consortium. One way of explaining its programs is via this method of swimlanes for people who haven't got time to look at the detail.

CSP, Communicating Sequential Processes, can be refined within itself. And so we can go on.

Let's go back just one here. What else have we got? Interestingly, take the second one down here. Game theory can be used to explain the intelligent behaviour of intelligent agents. Namely, how they negotiate perhaps, in order to be awarded the resource that they need if they're visiting some site where they're being questioned – I'm talking about software agents of course. Game theory is one of the things that can be used to explain their behaviour.

A logic of trust can be realised by authorisation protocols. I'll authorise you if I know somebody who says that you're okay or, ... of course it's more complex than that. I won't go into it any further because it's not my specialism.

I met a very interesting model with some colleagues the other day for ubiquitous computing: an engineering model of self-managed cells, where the behaviour of these cells, which are parts of a ubiquitous computing system say part of a healthcare system. They're self-managed in the sense that they know how to welcome new members, how to reject new members, how to understand the environment around them. The question is: how is their behaviour described? Well their behaviour is described informally at the moment. But I believe that some kind of a logical specification can be brought in to explain why that behaviour makes sense.

So there are plenty of things to think about.

We want to use these models to present existing knowledge and ongoing work. As we do this, we'll probably identify certain fields of Informatics as sub-towers, and within the field, the members will be related by explanation, and the field itself will be related by explanation outside itself. The challenge in my view is to identify different kinds of explanation. We have many examples - what do they have in common? I don't have the answer, but I believe we have enough hope that it's worth looking for an answer.

I talked about automating explanations. Software engineering must automate any explanation that's made precise by software sciences. So this, to me, is the relationship. Software

engineering is going to automate, is going to make feasible, the explanations that are made precise by the scientists. That's a grand challenge for ubiquitous computing. There are other grand challenges, one of which for example was to beat a grand master at chess. The question is: which kind of grand challenges is going to work more seriously for Informatics as a science? That one, I don't think is going to do it very well, but I think that to develop the organising principles for a science along the basis of models, or whatever scientific principles we evolve, that is the kind of thing that will justify the notion of a grand challenge. It is a grand challenge, and if we can get over the hurdles, the difficulties, then we shall be recognised as a science. In other words, the two kinds of challenge together will embed computing in our scientific culture.

I'm well aware that my talk I've based everything on the notion of model, and some attempt to explain what it is. I hope I've prepared the ground for Gilles' talk, where he's got a different challenge, and different perception of the way in which science might relate to the engineering. But that's end of my talk – thank you for listening.