The Importance of Logic in the Informatics Curriculum

Peter Bernard Ladkin
20140217

RVS White Paper 6

In Bielefeld, we teach Informatics (which others call Computer Science) to Bachelor's degree level in four degree courses: Informatics for the Natural Sciences (INS), Cognitive Informatics (CI), Bioinformatics and Genomics (BIG), and Media Informatics and Design (MID). Informatics for the Natural Sciences combines Informatics with studies in one of the natural sciences (Physics, Chemistry or Biology) or linguistics. Cognitive Informatics is aimed towards artificial cognitive functions in anthropomorphic robotics. Bioinformatics is, I hope, self-explanatory nowadays. Media Informatics is aimed at artists who wish to use new electronic devices for their work – this includes film and photography, since nowadays much work, editing and so forth, is mediated through digital-electronic HW and SW.

The coursework is divided into compulsory, constrained-elective, and purely elective (these are more or less anything throughout the university, but for limited credit).

Logic Courses in Bielefeld Informatics

Logic used to form part of a Theoretical Informatics course (which also taught Automata Theory and the Chomsky Hierarchy of Languages in the customary breadth). See below for details.

Boolean logic for circuit design is taught in a Computer Architecture course which is compulsory for the same degree courses as the Theoretical Informatics course. See below.

I developed additionally a two-semester elective module (two courses) in Applied Logic because it seemed to me, first, that our students were to my mind not adequately learning the customary features of formal syntax and formal semantics in formal languages in order to be able to use these as (I thought) customary in informatics. The first semester concentrates on propositional logic, in its Hilbert-style (axiomatic), Gentzen-style (both natural deduction and sequent-calculus) and Scott-style (sequent calculus as "consequence relation") formulations, and the exercises are formal inference and intermodal comparison, including with formally weaker logics such as intuitionistic propositional logic. The second semester introduces normal modal logics, their syntax, Kripke semantics and some uses, including temporal logic and a selection from such topics as BDI modelling of intentions, "gap" (many-valued, such as the Łukasiewicz Ln logics, or similar Kleene logics) and "glut" (paraconsistent) propositional logics and their technical relations, logics of decision and probability (useful for cooperative decision-making, and for decision-making in situations of uncertainty) and potentially others.

I have had philosophy students take this module; indeed, the first students to take it were all philosophers.

Besides the two core courses in Applied Logic, I offer seminars entitled Themes in Applied Logic Z (where Z is a Roman numeral currently between I and VI), which consider various logics, and aspects of logic, not already covered in the courses Applied Logic I and Applied Logic II. Few students have taken these seminars thus far.

Change in Degree Courses

German universities have changed in the 21st century to a Bachelor's/Master's model, dependent on accumulating Credit Points for individual courses, with per-course evaluation, to facilitate movement of students and transfer of academic achievement amongst universities throughout Europe. This is in general a good thing, but adaptation was hard, in particular because few German academics have hands-on experience with individual course evaluation and Bachelor/Master degree-course regimes and therefore lacked intuition for the advantages and pitfalls.

After our first pass at this, the Bielefeld Informatics part of the Technical Faculty organised a day retreat some years ago to discuss curricula. The curriculum of Theoretical Informatics was discussed, and it was agreed to cover just the automata and languages theory in the Theoretical Informatics course, and let the logic be covered by my Applied Logic module (or at least the first course of it).

Theoretical Informatics is required for two of our four Bachelor's degree courses, but somehow Applied Logic remains an elective for all, rather than compulsory. I recently asked the Dean and others how this came to be; people seemed to be unaware of the situation and I have found no one to give an answer as to how the decision was made.

The Issue, And How It Arose

I think **some understanding of the mechanisms of formal logic is essential to any Informatics degree curriculum.** Not all my colleagues agree. Indeed, having discussed it in one faculty meeting and one meeting of the curriculum committee, I would guess that nobody agrees. This note is an attempt to say why it is essential. I refer below to the assertion in bold font as the Thesis.

There is a reason why the question arises now. I was recently informed by a student working in my group that he had applied to another university in Germany well-known for its technical informatics, the Technical University of Braunschweig, for an Informatics Master's degree course, and been informed in a telephone call that he didn't appear to have enough logic to be admitted. He noted that he was currently taking my logic course, as well as one in the philosophy department, and was subsequently admitted provisional on successful completion. I thought it anomalous that we could be conferring students with a Bachelor's in Informatics who didn't have enough background in the fundamentals to continue study elsewhere. The majority of my colleagues don't appear to find this problematic – indeed, I appear to be alone in my view that this is anomalous. Discussions amongst my colleagues, though, as well as the information from my student, lead to the conclusion that study in logic appears to be the only systematic problem.

Before discussing the importance of logic, I briefly indicate our current curricula.

Summary of Bielefeld Informatics Compulsory Courses

Coursework is planned ideally over six semesters, although people do take longer. INS, CI and BIG study Algorithms and Data Structures using the programming language Haskell in the first semester, and continue with Object-Oriented Programming in Java in the second semester, meanwhile taking compulsory mathematics courses. MID is similar but has specially-adapted courses. All continue in the third and four semesters with a group SW project involving UML and Java, called “Techniques of Project Development”, as well as Databases. In addition, INS and CI study Theoretical Informatics, Algorithmics, Computer Architecture, Operating Systems, and take a Digital Electronics Lab, while BIG and MID study a condensed version of Architecture/OSs. BIG study Sequence Analysis and MID Human-Machine Interaction. CI study Foundations of Artificial Cognition. The fifth and sixth semesters are electives of various sorts, plus work on a short thesis.

Logic in Bielefeld Informatics Courses

Logic occurs in just three places in our compulsory courses.

First, a *very* brief overview is given in the Mathematics courses in the first two semesters for INS and CI (I don't know about BIG and MID). These courses are taught by the Mathematics Department, and I think it would be fair to say that no one who teaches it really knows of the applications of logic in Informatics, except for the traditional connection with digital logic. Students persistently tell me that they learn nothing about how to apply logic in their mathematics study.

Second, Boolean logic is taught as usual in the Computer Architecture course, taken by INS and CI. Karnaugh Diagrams are taught, as well as Quine-McCluskey minimisation, and the usual stuff. The connection with propositional logic is made, but not necessarily practiced. BIG and MID take a reduced course, but I don't know how much practice is required.

Third, the logic formerly taught by my colleagues in Theoretical Informatics (comprising the syntax of propositional logic, truth tables, the language of predicate logic) is now taught to CI in the compusory course Foundations of Artificial Cognition.

A Brief, Crude History of Logic in Computing

Historically, the formal mathematics of computable functions in the twentieth century is inextricably tied up with formal logic. The three formal models of computing proposed in the 1930's were Lambda Calculus (Church, a logician), the general recursive functions (Gödel, also a logician) and Turing Machines (Turing). Turing is primarily known for his Machines, his Test (for when an agent with whom one is interacting verbally can reasonably be regarded as "intelligent") and his codebreaking skills (in the war effort at UK's Bletchley Park). But he also worked in logic, amongst other things proving the undecidability of (classical) Predicate Logic. When I was a student in the 1970's, the theory of computable functions was considered part of Mathematical Logic. I took courses from Turing's PhD student Robin Gandy. Now, the theory of computable functions has expanded and takes place as much in Theoretical Computer Science.

Logic was in the foundations of Artificial Intelligence (AI), not only through a connection with Turing but also through the influence of John McCarthy, who not only turned the lambda calculus into a programming paradigm through his language LISP, but also formulated problems in AI amenable to a solution in applied propositional, predicate and even higher-order logic. Nowadays, the kinds of solutions to questions of artificial intelligence which arise from manipulating symbols which have some connection with a meaning are called Symbolic Artificial Intelligence.

At the end of the 1960's, Bob Floyd and Tony Hoare independently showed how one may verify that imperative programs actually did what one wanted them to do, through (now) so-called Floyd-Hoare Logic, in which a proof that a program in an imperative language did what one wanted it to do was built up incrementally from considerations of what each imperative statement in the program accomplished behaviorally. This notion of action was given by describing *before* and *after* states in the language of predicate logic, so-called *pre-* and *post-conditions*. Much imperative-program verification nowadays still uses an adapted Floyd-Hoare paradigm. Program verification of some sort is essential for ensuring the dependability of software. Temporal Logic, a variety of modal logic in which the modality generalises over time, has been used extensively to verify algorithms in concurrent computation, in which two or more processors (or agents) are operating at once and need to coordinate.

Again in the 1960's, Ted Codd at IBM in California invented a paradigm for data bases, very large collections of data with myriad properties ("attributes" in database-speak), called Relational Databases, along with a theoretical query language called Relational Algebra (not to be confused with Relation Algebra, which is a branch of Algebraic Logic). The theory of Relational Databases was widely extended by the logician Ron Fagin from the 1970's, and relational databases gradually became ubiquitous. Halpern et al. point out that the common (ubiquitous?) database query languages SQL and QBE are "s*yntactic variants of first-order logic*" http://works.bepress.com/neil_immerman/1/ .

Why Is Logic, Along With An Ability To Use It, Essential?

Logic has a variety of current applications in Informatics at any level.

First, an understanding and facility with formal languages of description (declarative languages, for short FDL) and their semantics (formal definitions of meaning) is required for specifying what SW is supposed to do, and indeed for checking that it has done it.

Second, defining and discovering what it is you want SW to do, and how to make sure it does all of what you want and none of what you don't want, nowadays involves various forms of simulation and modelling of the intended operating environment, discovering solutions to and constraints on issues which arise, and transforming those somehow, accurately, into requirements of software artifacts (and of course checking that the transformation is accurate). This is nowadays called "model-based development" (MBD). It is unthinkable to engage in MBD without some sort of facility with some FDL and its semantics, or indeed many such.

Third, understanding and using Relational Databases requires some sort of facility with query languages such as SQL and QBE, and as noted these are syntactic variants of the FDL Classical Predicate Logic.

Fourth, studies in artificial cognition require some sort of facility with dealing with intentions and intensional acts (the "s" is crucial!), such as that offered by BDI concepts. BDI is formally a modal logic; understanding and facility with such is therefore very helpful.

Fifth, it is widely recognised nowadays that electronic HW (microprocessor) designs has become too complex to be effectively checkable "by hand". Automated methods are essential, and those methods are exclusively based on formal logical languages, both for positive verification (in which a desired property is proved to hold from known properties of a design) and for model-checking (in which an undesired property is shown to be absent through enumerating the possibilities in which it might be present and showing it isn't). HW design checking is a complex, specialist process whose technology goes way beyond that teachable in an undergraduate informatics course, but it is an essential part of informatics, even outside of chip manufacturers. See Section 6 of Halpern et al. (*op. cit.*).

Applications to Curriculum

It is generally regarded that an ability to design and write SW which is somehow fit for its intended purpose is one of the achievements of pursuing a degree course in Informatics successfully. As indicated above, logic has been successfully applied to ensure dependability. The question here is whether some facility with logic is essential to such endeavor.

It is generally thought nowadays that a facility with handling databases requires skill with query languages such as SQL and QBE. Since these are syntactic variants of FOL, any skills with FOL

should transfer directly. One must, however, be aware of a possible translation problem, such as that between circuit logic (Boolean logic) and propositional logic, and thereby make explicit effort to facilitate easy translation.

Although verification of electronic HW (microprocessor) designs involves automated or semi-automated methods involving facility with formal logical-language processing, it could be aguedd that such design checking is a complex, specialist process whose technology goes way beyond that teachable in an undergraduate informatics course. However, one can also note that it is an essential part of modern informatics, even outside of chip manufacturers. Lower-complexity FPGA designs with dependability requirements (most of them) must also be verified, so one must use the available techniques, which all involve facility with use of a formal machine-manipulable description language with an unambiguous semantics.

Software Dependability Assurance, Specifically

In the course of a project administered by the German electrotechnical standards organisation DKE on behalf of the German Federal Ministry for the Economy and Technology (BMWi) under the INS program, the author and Bernd Sieker have inquired into the necessity of various techniques for assuring the dependability of SW (Project INS 1234). The general theme of dependable SW and how to enhance dependability has been addressed by a US National Academy of Engineering inquiry, chaired by Professor Daniel Jackson, which reported in 2007 (Software for Dependable Systems: Sufficient Evidence? ed. Daniel Jackson, Martyn Thomas and Lynette I. Millett, Washington, D.C., National Academies Press, 2007).

In the first part of the project INS 1234, I asked Martyn Thomas and Michael Jackson, originator of the highly-praised industrial SW development methods Jackson Structured Programming (JSP) and Jackson System Development (JSD) as well as the Problem Frames methods for SW requirements engineering, about crucial aspects of requirements specification and analysis for safety-critical software systems in general (safety is part of dependability according to the IFIP classification but not according to the IEC classification). [1]

Both Thomas and Jackson suggest that checking requirements specifications for consistency is essential. In their experiences, requirements for moderately complex software systems are moderately prone to be inconsistent (the reason appears to be that various domain experts are required to help formulate system requirements, and the domain experts themselves remain unaware that properties they need may conflict with properties required by other domain experts. It requires a specialist in requirements engineering to help formulate all requirements in a uniform language which then may be manipulated to check for consistency). Consistency is evidently a notion of logic, and is meant here in exactly this way. Both Thomas and Jackson support formulation of

---

1. Safety is part of dependability according to the IFIP classification but not according to the IEC classification. For the IFIP WG 10.4 definitions, see Dependability: Basic Concepts and Terminology, ed. J. C. Laprie, Springer-Verlag Wien, 1992, and Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C., Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, vol. 1, pp. 11-33, 2004. For the IEC definition, see http://tc56.iec.ch/about/faq.htm . In particular, Section 5 of the FAQ explains that safety and security are not considered to be within scope. Definitions developed are in www.electropedia.org Part 191: Dependability and Quality of Service, but this section does not include a definition of "dependability" itself. The author understands that IEC definitions in the safety and security areas will eventually appear here, though. The author can speak from personal experience that the relation between the work of IEC Technical Committee, 56, which has responsibility for dependability, and those such as SC 65A concerned with E/E/PE system safety and security remains to be clarified.

requirements in some formal language with an unambiguous semantics, in order inter alia to allow consistency checking. This may proceed by hand, but preferably automatically or semi-automatically to enhance the likelihood of correctness of the result. Such automatic checking may evidently only be performed on expressions in a formal language. (See Peter Bernard Ladkin and Bernd Sieker, Practical Formal Methods, work in progress under INS 1234, to appear 2014.)

There is also a need for checking whether requirements are (in the terminology preferred by the author) relatively complete. Although there is a logical notion of completeness, indeed the author has defined a such a notion applicable to system requirements, there are other notions of completeness which are not logical. Although ideally one would wish to check all such criteria, it is widely regarded as impractical to do so. Hence assessing relative completeness of requirements is not invariably associated with a facility with FOL or other logics. (See Ladkin & Sieker, *op. cit.*)

Numerous studies have shown that failures of dependability in dependability-critical SW may be assigned in a proportion of between 70% to over 90% to problems with the actual operational environment of the software not being thoroughly covered by the specification of the SW requirements, or other problems with the requirements (such as inconsistency). It follows that the adequate analysis of requirements is central to the enhancement of dependability of SW, and we have seen that a facility with a formal language with an unambiguous semantics, as well as tools to check logical properties, are essential skills for requirements analysis.

Most lay people seem to think that people with a university degree which includes informatics should be able to write computer programs, and that the computer programs they write should do what is intended, and should not cause harm. “What is intended” is captured by a requirements specification. Doing it is the property of reliability, part of dependability. Not causing harm is the property of safety, part of dependability (in the IFIP construal). We have just seen that a facility with logical languages is necessary to analyse requirements specifications for dependability properties. It follows that a facility with logical languages sufficient for this purpose is a skill which most lay people would expect informatics graduates to possess.

I consider the consideration in the preceding paragraph to be sufficient, alone, to establish the Thesis. The necessity of skill with syntactic variants of FOL for any study of contemporary database engineering is another observation in favor of the Thesis.

For those who wish to go in to industry, a facility in logic for both SW and HW dependability engineering seems unavoidable. For those who wish to pursue work in artificially cognitive systems, such as anthropomorphic robotics, some facility with modal logics sufficient to work with models of intention such as BDI currently seems at least very important if not necessary.

Further Work

A series of affidavits from system-dependability practitioners concerning their experience and the importance of logical understanding and associated techniques to their work is being prepared.

Other selected university curricula show a clear requirement for facility with logic for success in the degree program. A summary of selected curricula is being prepared.

Acknowledgements: Many thanks to Gergely Buday, Les Chambers, Dewi Daniels, Patrick Graydon, Alvery Grazebrook, Michael Jackson, Hauke Kaufhold, John Knight, Philip Koopman, Tobias Nipkow, Bertrand Ricque, Tim Schürmann, Bernd Sieker, Michael Tempest, Martyn Thomas, Steve Tockey and Nick Tudor for discussion, some of which will appear in future work.