

Computing Theory With Relevance

Wayne Brookes

Faculty of Information Technology
University of Technology, Sydney,
PO Box 123, Broadway, NSW 2007,
Email: brookes@it.uts.edu.au

Abstract

In computer science education, the topic of computing theory is one that is commonly not well received by students. Career-oriented students often view the topic as irrelevant, and would rather learn new skills and technologies that they perceive will improve their future employment prospects. This paper outlines an approach that attempts to blend these two apparent extremes by using “popular” technologies, including XML, to motivate and illustrate concepts of computing theory in a first-year undergraduate computing subject.

Keywords: computer science education, computing theory, XML

1 Introduction

Computing theory encompasses a range of topics that address the mathematical foundations of computer science. Typically it includes topics from discrete mathematics, algorithmic complexity and computability, finite state automata and regular expressions, and others.

Teaching computing theory topics at undergraduate level has always been a challenge. Students often struggle with theoretical topics; educators search for new ways to make theory seem interesting and relevant; and some graduates working in industry look back and question the usefulness of ever learning computing theory at all.

There are pressures from students and industry to reduce the amount of theory covered and instead offer more practical skills. This has become increasingly so with the diversification of computing degrees from traditional “computer science” to include more marketable degree titles such as information technology, information systems, software engineering, Internet computing and electronic commerce.

This reduction in theory coverage is discussed by Tucker et al. (2001) in some detail. Tucker et al. argue that “Our discipline is thus presented as less like a science and more like a collection of techniques and artifacts that reflect current technologies. This strategy may prepare graduates for today’s technology, but it will likely not prepare them well for the longer haul.”

On the relevance of computing theory to computer science education, the Joint IEEE Computer Society / ACM Task Force on Computing Curricula 2001 (IEEE Computer Society and ACM 2001) emphasises the importance of various elements of computing theory to undergraduate computing curricula, particularly discrete mathematics. Indeed the Discrete Structures area in *Computing Curricula 2001* contains more core hours than any other

knowledge area. Despite this, *Computing Curricula 2001* has also been criticised for its reduction in computing theory topics since the previous version released in 1991 (Tucker, Kelemen & Bruce 2001).

While there seems to be some agreement on the importance of various computing theory topics in the computer science education community, there is also acknowledgement that these topics are often conceptually difficult for students, and often difficult for students to see the relevance to their future careers.

This paper presents one approach to try and address the problem of making computing theory relevant for students. Traditionally, computing theory is taught in a separate subject, often in the second or third year of the degree. Rather than teaching computing theory as a standalone subject, here it is integrated into a larger subject that covers a range of foundation issues on distributed and Internet-based computing. Relevance is provided by linking computing theory topics with XML (Extensible Markup Language) and related practical topics. On the surface, the students may think they are just going to learn XML. In practice, they also learn some of the theoretical underpinnings of XML, and indeed some of the theoretical underpinnings of computer science in general.

While this does not reduce the conceptual difficulty that the students face, it does offer additional motivation for students to study computing theory topics that are often otherwise perceived as dry and irrelevant. This paper argues that linking computing theory with currently popular technologies can be an effective approach of integrating theory into a practice-based undergraduate computing curriculum.

The remainder of this paper is organised as follows. Section 2 presents an overview of the subject that was run, both its content and its overall positioning within the degree program, and section 3 describes the subject’s learning modes and assessment pattern. Section 4 describes in detail how computing theory topics were linked with XML. Section 5 considers the benefits and challenges of the approach, and section 6 examines other authors’ approaches to improving students’ learning of computing theory topics.

2 Background

This section introduces the background of the subject and the motivation for the presented approach of integrating computing theory topics with XML technologies.

2.1 Content

The first important point to note is that the subject is not about computing theory, but rather focuses on distributed computing. The unifying theme of the subject is how various technologies and techniques can be used to address problems in the application domain of electronic business (e-business).

Approximately half of the subject covers topics from distributed computing, including client/server, three-tier and N-tier distributed architectures; network communication mechanisms such as sockets, Remote Procedure Call, and middleware technologies; and an introduction to how such technologies fit together in the e-business domain. The other half of the subject covers XML and related topics, such as XML Schema, XSLT, and the DOM and SAX parsers for XML. These are explained in more detail later in the paper.

At first glance, this seems an unlikely place to cover aspects of computing theory. However it forms part of the overall design for the undergraduate curriculum at the University of Technology, Sydney (UTS). The previous version of the curriculum included fairly traditional, standalone subjects on computing theory: one in first year and one in the final year. Feedback from students and graduates indicated that the standalone theory subjects were perceived as not being relevant, and that the subjects seemed out of line with the practice-based focus of the remainder of the course.

When the curriculum was redesigned, a decision was made to remove the standalone theory subjects, and instead to integrate aspects of computing theory into other core subjects in the curriculum where appropriate. Thus computing theory becomes a constantly recurring theme throughout the degree program, and rather than being taught as a single unit, instead is taught in the context of other topics where relevance can be demonstrated. In this approach, every core subject becomes a potential candidate for inclusion of some aspects of computing theory, if relevance can be demonstrated. Computing theory is not the only topic to be treated as a recurring theme throughout the degree program. A similar approach is taken in covering material on ethics, usability, business needs and generic skills.

In the context of the subject discussed here, an important implication is that the distributed computing topics drive the computing theory, rather than the other way around. Traditional subjects on computing theory focus on the theory first, and then present examples of where it may be used. In the approach described here, the application domain is introduced first, and the computing theory is presented as the foundation.

A danger in this approach is that students may not see computing theory as a single unified branch of computer science. Different universities generally have different approaches to teaching computer science. At some universities, theoretical foundations of computer science strongly drive the curriculum, while at others, current business needs have a stronger influence over the curriculum. The University of Technology, Sydney leans more towards the business-oriented end of the spectrum. The approach presented here may not fit well into the overall curriculum design at universities more strongly driven by theoretical foundations. However in more theoretically-oriented universities, the approach presented here may possibly be used to augment (rather than replace) standalone subjects on computing theory.

2.2 Context

The subject is a compulsory subject in the university's undergraduate information technology degree programs, and is typically studied in the second semester of first year. This has three implications. Firstly, the enrolment is large, with up to 350 students enrolled at one time. Secondly, as the subject is compulsory, it includes students from diverse backgrounds, and who have diverse career aspirations. This also means that not all students enrolled in the subject are interested in the subject content. Thirdly, as it is a first-year subject, the content covered must be appropriate for that level.

In particular, in their first semester at university,

students would have completed one subject on object-oriented programming; one subject introducing distributed and Internet-based computing, and computer architecture; one subject on introductory computer networking; and one subject introducing information systems. This restricts the amount and kinds of computing theory that are appropriate for this level.

3 Approach

The subject includes a fairly typical pattern of lectures, tutorials and laboratory sessions. However, rather than the traditional approach of having tutorials and laboratory sessions based principally on material presented in the lecture, instead each type of session approaches the content from a different angle.

The laboratory sessions provide a series of exercises that students work on involving XML and related technologies. The XML content is primarily delivered via the laboratory sessions and associated readings. Very little time is spent in lectures or tutorials on practical matters like language syntax.

The lectures focus primarily on conceptual material, making references to the XML languages when necessary. For example, when XML is first introduced, the lecture content is actually about trees. XML is used as an example of a tree-structured document format, and XML examples are used throughout the lecture, but the XML syntax is not the focus of the lecture.

The tutorials include some exercises based on the lecture material. But as an added dimension, the tutorials also include a small number of student presentations of articles broadly related to that week's topic. For example, when the week's topic is finite state automata, the articles for that week cover how automata are used in web applications, and how automata are used in UML (Unified Modelling Language) modelling.

The assessment for the subject consists of a mid-semester examination covering only theory topics, a final examination covering a mixture of theoretical and practical topics, a practical assignment where students create a simple web-based application involving XML, and some marks are also awarded for students' tutorial presentations and contribution to class discussions.

4 Linking XML with computing theory

As already discussed, the approach taken was to define the subject content in terms of distributed computing topics (in particular, XML) and to use computing theory to support their presentation. This section illustrates how some of the XML technologies were linked with associated computing theory topics.

4.1 XML and trees

XML is a language for structured information representation. It is a tag-based markup language, and the tags are nested resulting in a hierarchical information structure. Thus it is quite natural to introduce tree structures when introducing XML.

Informally, tree structures are already familiar to most students, for example from their experiences with hierarchical file systems.

What is new for students is approaching trees in a formal manner. The subject introduces the mathematical definition of trees as a set of vertices and a set of edges, with all the vertices connected, and without any cycles in the set of edges. Examples of XML documents are used to illustrate the basic tree concepts.

Tree traversals are also introduced: preorder, inorder and postorder. This is very relevant for XML, as most programs that process XML documents need to decide in

which order they will process each node. Books and articles introducing XML for programmers often mention the XML “tree walking algorithm”. In fact, this tree walking algorithm is just a preorder traversal.

Using simple trees as a starting point, more complex kinds of trees (such as binary search trees) are introduced to students, even though they are not directly related to XML. Some coverage of graphs is also provided, although again this is not directly related to XML.

Overall, XML and trees are a good match of topics. XML itself is a simple language, and easy for first year students to understand. Trees are also a concept that students easily grasp informally. Presenting a mathematical introduction to trees as part of a module on XML is an ideal match of theory with practice.

4.2 DOM and SAX parsing

Many students start with the myth that XML is a replacement for HTML, and the primary use of XML is for creating web pages. This subject introduces broader uses of XML, in particular examining the use of XML as a language for networked applications to exchange information. In order for an application to use an XML document, it needs to read the contents of a document stored on the filesystem and store the document’s data into variables in a running program. This introduces the need for parsing.

Document Object Model (DOM) and Simple API for XML (SAX) are two standardised programming interfaces for parsing an XML document. A DOM parser reads an entire XML document at once, and constructs a tree representation of the document in memory. The DOM API then provides various methods for accessing different parts of the in-memory tree structure.

A SAX parser is event-driven. A SAX parser reads an XML document sequentially, and triggers events when various kinds of XML markup are encountered. The programmer is required to implement an event handler that is invoked when the events are triggered.

DOM and SAX are language independent interfaces for XML parsing. Implementations of DOM and SAX parsers are available for a range of different programming languages. In this subject, Java implementations of the parsers are used, as the students are already familiar with Java from their prior studies.

Many textbooks and technical articles introducing DOM and SAX focus on providing cut-and-paste examples of source code to use the parsers. However in this subject, where the goal is to introduce elements of computing theory in context, DOM and SAX provide an opportunity to introduce the theoretical concepts of formal languages, grammars and parsing.

Thus, before discussing DOM and SAX, the lecture stream of the subject introduces the concept of formal languages. The parsing process is described, including the construction of parse trees and abstract syntax trees, leveraging off the students existing knowledge of trees. In discussing how a parser takes an input string of characters and constructs a parse tree, the topic of grammars naturally arises. The mathematical definition of a context-free grammar is introduced, as well as the Backus-Naur Form (BNF) notation for writing grammars. The grammars discussed in this part of the subject are all context-free, however regular grammars are also covered in a later section of the material.

Finally, the DOM and SAX parsers for XML are introduced. Students find the operation of the DOM parser relatively easy to understand, as it fits with their existing notion of tree-structured XML. However the SAX parser is not so intuitive as students have had limited exposure to event-driven programming at this stage in their degree.

In summary, discussion of how to use information from an XML document in an application provides a convenient way to introduce the concept of parsing. In this subject,

the laboratory sessions provide an opportunity for students to use existing parser implementations to read XML documents into their own applications, while the lectures provide a forum for presentation of some of the theoretical background to formal languages, grammars and parsing.

4.3 XML Schema and Regular Expressions

XML Schema is a mechanism for defining restricted subsets of XML that are appropriate for particular usage scenarios. Each schema defines a set of rules. An XML document instance that conforms to these rules is said to be *valid* (or *schema-valid*). For example, in a business context, there might be one XML Schema that defines the format of an XML purchase order, and another XML Schema that defines the format of an XML invoice. The schema definition includes at least the set of elements (tags) permitted, their sequencing, and the values that are permitted for certain elements.

The XML Schema specification includes a rich set of data types that element values may have, including integers, decimal numbers, date and time values, strings, etc. It also provides rules for creating constrained subsets of these data types, such as specifying the maximum and minimum values for numbers, specifying the length of a string, etc. When the data type of an element is a string, it can also be constrained by using a regular expression.

Regular expressions lead nicely into the discussion of regular languages and regular grammars. From a theoretical perspective, this complements the earlier presentation of context-free languages and context-free grammars.

In traditional computing theory subjects, discussion of regular expressions often leads into the topic of finite state automata, and the same approach has been taken here. Finite state automata are introduced as “machines” that can implement a regular expression. Automata are presented both diagrammatically and mathematically.

Unfortunately it is not as easy to use XML or XML Schema to motivate discussion of finite state automata, as automata have little obvious direct relationship with XML or XML Schema. Fortunately there are other ways to motivate discussion of finite state automata, such as statechart diagrams from the Unified Modelling Language (UML), and Model-View-Controller design patterns where the controller is often an implementation of a finite state automata.

Overall, students respond well to the theoretical material on regular expressions, and can clearly see the relationship to XML Schema and its use of regular expressions. However it is a little harder to convince students of the relevance of finite state automata in this context. As this is the students’ first formal exposure to automata, and the fact that the material is somewhat more abstract than regular expressions (which students can experiment with in laboratory sessions), it is not surprising that students struggle a little more with automata concepts.

4.4 XSLT and Declarative Programming

The final example of using XML technologies to illustrate more theoretical aspects of computing is the use of XSLT (XML Style Language: Transformations) to illustrate the concept of declarative programming. This example is perhaps less theoretical than the previous three, as it does not rely upon mathematics as its foundation.

XSLT is a language that allows the definition of a set of rules that can be used to transform one XML document into another format, often another XML document with a different structure. For example, XSLT can be used to rename elements (tags) in an XML document, or can be used to selectively extract a subsection of an XML document. Another common application of XSLT is to convert an XML document into an HTML document suitable for display in a web browser.

In most computing curricula, a number of different programming languages are presented in core subjects. Typically this includes two or more object-oriented and/or procedural (imperative) languages as well as at least one other language that is of a quite different style. Often the non-imperative language introduces functional programming or logic programming, with the intention of challenging students' understanding of programming language concepts.

In this subject, rather than present a functional or logic-based language, XSLT provides the opportunity to use declarative programming as an alternative to imperative approaches.

An XSLT "program" consists of a set of templates (rules) that each may match a certain part of the input XML document (e.g. a particular element). When a match is encountered, the XSLT template is invoked to generate output depending on the contents of the template. The templates are not invoked sequentially, but rather are invoked whenever a matching element is encountered in the input. One template may be invoked many times, or a template may not be invoked at all, depending upon the input XML document.

This challenges students' views of programming, as until this point in their degree, the principal programming language encountered is Java. Indeed when students begin to write XSLT, their first instincts are to think of writing XSLT templates in sequential steps.

This example of using XSLT to illustrate declarative programming is not quite as theoretical as the previous examples presented, but offers another way to leverage XML related technologies to introduce more abstract computing concepts.

5 Evaluation

The approach described in this paper aims to better demonstrate to students the relevance of various computing theory topics by presenting them as foundational material to XML. Overall, based on student feedback, the approach seems to have been successful.

The XML and computing theory topics were covered in the first half of the subject, and the mid-semester examination was the main assessment instrument for the computing theory topics. Shortly after the mid-semester examination, students were asked to complete an anonymous survey about their experiences in the mid-semester examination.

One of the questions was "Please rate how well you think the theory covered in the subject relates to practice in the real world?". Answers were on a five point scale: 'always', 'usually', 'about half the time', 'occasionally' and 'never'. Of 109 students enrolled, 50 students responded to the survey. Of those who responded, 26% of students indicated that they could 'always' see the relationship between theory and applications in the real world. Another 32% chose 'usually' (total 58%). Of the remainder, 32% said 'about half the time', 10% said 'occasionally', and no students chose 'never'.

It is worth noting that this question was part of a six-question survey, and was surrounded by other questions about the difficulty of the exam and the topics on the exam. This was done so as not to draw special attention to this particular question.

This survey indicated that 90% of the students who responded could see the relationship of theory to practice 'about half the time' or more, with 58% seeing the relationship 'usually' or 'always'. Although I do not have data to compare this with a more traditional approach to teaching computing theory, these results are encouraging.

Anecdotal feedback from students throughout the semester indicated that although some found the theoretical material challenging, they were generally positive about how the theory was linked to XML. It would also be

reasonable to say that the majority of the students seemed keenly interested in XML.

Analysing the students' performance in the final examination, overall performance was about the same for both the theory-oriented questions and the more practically-oriented questions, hopefully indicating that students divided their attention appropriately between the XML and theory topics.

My own reflection was that overall the approach seemed successful. Creating XML-oriented examples that could be related to the otherwise symbolic world of computing theory certainly made the development of material both more challenging and more interesting.

On the negative side, the unusual combination of topics meant that a single suitable textbook could not be found for the subject. Instead, students were referred to chapters from different textbooks as appropriate, as well as other background readings. Students were also provided with detailed lecture notes to support their learning.

Another point to note in evaluating the approach taken is that the subject only covered those topics in computing theory that could be reasonably related to XML technologies. This was in keeping with the approach of teaching computing theory in the context in which it can be demonstrated as most relevant. This meant omission of some computing theory topics that would normally be found in a more traditional subject on computing theory. Some topics were also covered more lightly than normally encountered in a traditional computing theory subject, either because the students are only in first year, or because of time constraints.

Some examples of topics that might traditionally be taught that were omitted include proof-by-induction techniques; nondeterministic pushdown automata as a way of recognising context-free languages; the Pumping Lemma for finite state automata; complexity and computability; and Turing machines. Some of these topics will be covered by later subjects in the degree program.

6 Related work

This paper presents an approach to making computing theory relevant by relating it to a currently popular technology like XML. This is not the only subject to have attempted this. A linkage between computing theory and XML has also been used in the teaching of a subject entitled "Languages and Algorithms" at Napier University in the UK (Napier University 2003).

However, the Napier approach differs from the one presented here in a number of ways. Firstly, the audience of the Napier subject is honours-level students in a Bachelor of Engineering (Honours) program, whereas the approach presented here is for first-year undergraduate information technology students. The Napier approach presents the theoretical material in more depth, as appropriate for honours-level students. Another significant difference in the approach is that the Napier subject is based on a theoretical topic — languages and algorithms — and XML is used as an example. In the approach presented here, the basis of the subject is actually distributed computing, and the theoretical topics are the supporting material, rather than the principal focus of the subject.

Other authors have considered the role of XML in the curriculum. For example, Paxton (2001) discusses the role of XML in the computer science curriculum. In particular, he talks about how XML might be integrated into the core computing curriculum, including suggestions such as (Paxton 2001):

- "In an advanced data structures and algorithms course, XML and DTD can be examined from the standpoint of the multiway tree abstract data type. Trees are a topic that need to be covered and XML provides a convenient mechanism to illustrate some of the concepts."

- “In a programming languages class, DTDs can be used as a practical tool for learning more about the Backus-Naur Form (BNF).”

In the approach presented here, techniques similar to these have been adopted (although XML Schema was used rather than DTDs because of its stronger data types and use of regular expressions), and other techniques have been added. However in the approach presented here, the computing theory supports the XML, rather than vice versa as proposed by Paxton.

Moving away from discussion of XML, other authors have taken various alternative approaches to making computing theory more accessible to students.

Perhaps the most common approach is to provide animation and simulation tools for supporting students’ learning of computing theory, particularly automata theory. Chesñevar (2003) presents a summary of some of the more common tools, although many such tool sets are available.

The approach of using animation and simulation tools to assist students in visualising some of the more abstract concepts in computing theory is complementary to the approach presented here. Indeed when the subject was run, students were provided with access to freely available simulation tools on the Internet to assist their learning.

Another common approach is to provide students with access to small executable tools that can be used to motivate and reinforce students’ learning of different theory topics. One example is the use of the Unix ‘grep’ tool to assist understanding of regular expressions. Another example is in learning algorithmic complexity, where students are often provided with (or expected to produce) different algorithm implementations and experiment with tools to analyse algorithm performance (for example, to illustrate the different complexities of recursive versus iterative algorithms).

Another successful approach to better supporting students’ learning of computing theory topics is to adopt a problem-based learning model (Hamilton, Harland & Padgham 2003). In this approach, students work in small teams on problems that require application of computing theory knowledge to solve. The tutorial classes become the main learning forum, and the lectures are providing the necessary background information for the problem solving process. While such an approach was not used in the subject described in this paper, the idea has some similarities in that the computing theory topics are being motivated by real problems, rather than the artificial mini-problems often presented in textbooks from which it can be difficult to see overall relevance.

Obviously the basic approach presented is not limited to just linking XML with computing theory topics. Other ‘popular’ technologies could also be chosen as a starting point to examine some underlying computing theory that is relevant. This is challenging and requires some creativity, as it goes against the way most textbooks will teach standard computer science topics. Beaubouef (2002) presents some other suggestions for where mathematics can be linked into the computer science curriculum with the most relevance.

7 Conclusion

This paper presented one approach to improving students’ perceptions of relevance of computing theory topics. Past experience shows that when taught in a standalone subject on computing theory, students often struggle with seeing the relevance of the material presented, even when many good examples are shown.

The approach taken here was rather than teaching a subject devoted to computing theory, instead to encourage students to learn specific aspects of computing theory in the context in which their relevance can best be

demonstrated. Specifically, this paper examined how various theoretical topics can be linked with XML and related languages.

XML is a currently popular technology, and as such, students perceive it as being highly relevant to their career prospects. Presenting some common computing theory topics as foundations to XML appears to have made the theory more relevant to students.

A challenge is to apply this method to topics other than XML. At the time of writing, XML is not often taught as part of a core undergraduate computing curriculum. However, the general principle of using a technology to motivate computing theory, rather than the reverse approach can no doubt be applied to other technical topics such as programming, operating systems concepts, etc. The challenge is in choosing topics that students will find interesting and perceive as relevant for their future careers. Unfortunately for educators, the technical topics students are most likely to find interesting and relevant are topics that are relatively new and/or transient, leading to difficult decisions about what to include in the curriculum.

Although the approach presented here is not the only way of making computing theory topics more accessible to students, the approach seems to have been successful, and provides another resource for the computer science educator’s toolbox.

Acknowledgements

Some of the computing theory lecture material for this subject was originally prepared by my colleagues at the University of Technology, Sydney, Dr Andrew Solomon and Mr Richard Raban.

References

- Beaubouef, T. (2002), ‘Why computer science students need math’, *ACM SIGCSE Bulletin* **34**(4), 57–59.
- Chesñevar, C. I., Cobo, M. L. & Yurcik, W. (2003), ‘Using theoretical computer simulators for formal languages and automata theory’, *ACM SIGCSE Bulletin* **35**(2), 33–37.
- Hamilton, M., Harland, J. & Padgham, L. (2003), Experiences in teaching computing theory via aspects of problem-based learning, in T. Greening & R. Lister, eds, ‘Proc. Fifth Australasian Computing Education Conference (ACE2003), Adelaide, Australia’, number 20 in ‘Conferences in Research and Practice in Information Technology’, Australian Computer Society, pp. 207–211.
- IEEE Computer Society and ACM (2001), *Computing Curricula 2001: Computer Science*, Final Report (December 15, 2001).
URL: <http://www.acm.org/sigcse/cc2001>
- Napier University (2003), ‘CO42010: Languages and Algorithms’, [Internet].
URL: <http://www.soc.napier.ac.uk/module/op/onemodule/moduleid/CO42010>
- Paxton, J. (2001), ‘XML in the CS curriculum: pointers and pitfalls’, *The Journal of Computing in Small Colleges* **17**(2), 106–111.
- Tucker, A. B., Kelemen, C. F. & Bruce, K. B. (2001), Our curriculum has become math-phobic!, in ‘Proc. 32nd SIGCSE technical symposium on Computer Science Education, SIGCSE 2001’, Charlotte, NC, USA, ACM Press, pp. 243–247.