Developing Computer Science Education - How Can It Be Done?

Workshop
March 10, 2006

IDA
Institutionen för datavetenskap
Linköpings universitet

CeTUSS
Centrum för Teknikutbildning i Studenternas Sammanhang

Workshop "Developing Computer Science Education - How Can It Be Done?"
Friday March 10, 2006,

9:30-10:40	**Session 1: Plenary Opening**	Session chair: *Anders Berglund*
9:30-9:40	Welcome Address	Prof. *Mariam Kamkar*, Head of Department, Department of Computer and Information Science, Linköpings universitet
9:40-10:40	Invited Talk: Research and Development in Computing Education	Prof. *Lauri Malmi*, Helsinki University of Technology
10:40-11:00	**Coffee in "Ljusgården"**	
11:00-12:00	**Session 2: Course Development I**	Session chairs: *Anders Haraldsson, John Wilander*
	2.1: Discussing programming assignments in a virtual learning environment	*Peter Dalenius*, Department of Computer and Information Science, Linköpings universitet
	2.2: Generalizations later: A bottom up course design	*Olle Willén*, Department of Computer and Information Science, Linköpings universitet
	2.3: Towards a Set of eXtreme Teaching Practices	*Roy Andersson, Lars Bendix*, Department of Computer Science, Lund Institute of Technology
12:00-13:00	**Lunch in "Ljusgården"**	
13:00-14:20	**Session 3: Tools for Computer Science Education**	Session chair: *Juha Takkinen*
	3.1: Authentic Examination System	*Torbjörn Jonsson*, Department of Computer and Information Science, Linköpings universitet
	3.2: Web-based UNIX training for first year students	*Peter Dalenius*, Department of Computer and Information Science, Linköpings universitet
	3.3: OMNotebook - Interactive Book Software for Teaching Programming	*Anders Fernström, Ingemar Axelsson, Peter Fritzson, Anders Sandholm, Adrian Pop*, Department of Computer and Information Science, Linköpings universitet
	3.4: Dealing with tasks in a realistic object-oriented system- The first result: understandings of the interface concept in Java	*Jonas Boustedt*, Department of Information Technology, Uppsala University
14:20-14:40	**Session 4: Tool Demonstration (and coffee in "Ljusgården")** x parallel tracks.	Session chair: *Olle Willén*
	4.1. Authentic Examination System (Paper 3.1), in a computer lab.	*Torbjörn Jonsson*, Department of Computer and Information Science, Linköpings universitet
14:40-15:40	**Session 5: Course Development II**	Session chairs: *Simin Nadjm-Tehrani, Johan Åberg*
	5.1: Teaching parallel programming early	*Christoph Kessler*, Department of Computer and Information Science, Linköpings universitet
	5.2: How to Construct Small Student Groups?	*Tim Heyer*, Karlstad University
	5.3: An Analysis of Gender Impact on Students' Performance in a Written Examination of Software Engineering	*Kristian Sandahl*, Department of Computer and Information Science, Linköpings universitet
15:40-16:00	**Closing Discussion**	*Arnold Pears*

Research and Development in Computing Education

Lauri Malmi,

<lma@cs.hut.fi>

Professor of Computer Science

Helsinki University of Technology.

Developing education is an essential part of the work of all teachers. Many innovative approaches, for example, in introductory programming education have been demonstrated by good teachers. However, in too many cases such novel ideas, tools or support materials are poorly disseminated among other teachers even in the same institute. The reason may the lack of evidence. Can we demonstrate actual progress in terms of better learning results, motivation or use of resources in our courses? This talk will discuss the relation of developing education and researching education. What can we gain from research when developing our courses?

Lauri Malmi is a professor of Computer Science at the Helsinki University of Technology. His field is computing education research, where he has been working on automatic assessment of programming and algorithmic exercises, interactive visual methods in learning algorithms, and problem-based learning in introductory programming courses.

Discussing programming assignments in a virtual learning environment

Peter Dalenius
petda@ida.liu.se
Department of Computer and Information Science, Linköpings universitet
2006-02-20

One important aspect of our introductory programming courses has been to cultivate the students' ability to reason about code. A programmer should not only be able to solve problems and design computational processes, but also to participate in a critical discussion about alternative solutions. The GENIUS[1] project allowed us to move that discussion into a virtual classroom.

The GENIUS project was funded by the European Commission and connected nine European universities between 2001 and 2003. The aim was to develop and evaluate learning environments that would enable students to participate in courses from other universities. Our contribution was to develop a small course in C++ programming that was tested twice, the second time with some students from other countries. The course consisted of 5 seminars conducted in a virtual classroom. Before each seminar the students were required to solve a few exercises and hand them in. Each student was then given another student's solutions to present and criticize during the seminar. The group consisted of 8 volunteer students in their second or third year.

We found that the virtual classroom, despite being a new environment both for students and teachers, supported the critical discussion. The ability to share applications, to point and click and even modify code as you were speaking, made the environment seem "surprisingly natural" according to the students.

Motivated by the positive response from the students, we decided to integrate the virtual classroom discussions into our ordinary courses. During 2004 and 2005 we moved from the integrated commercial platform used in the GENIUS project to an environment that was a mix of different open source tools. A group of volunteers from our first year programming course participated in 3-5 virtual classroom seminars with approximately the same contents as our real classroom lessons. These students found the experiments with the online seminars interesting, but were considerably less motivated to participate than our C++ students.

I believe that three main factors lie behind successful use of online seminars: a) Both teachers and students need some *training in using the technology* so that it does not stand in the way of learning. b) The teacher has to be aware that *the virtual classroom pedagogy* differs from ordinary classroom pedagogy. For example, the teacher sometimes need to talk the students through the seminar to make up for the lack of visual queues. c) The course has to be organized so that the students are *stimulated to discuss problems* during the seminars instead of just silently participating.

[1] Generic E-Learning Environments and Paradigms for the New Pan-European Information and Communication Technologies Curricula

Generalizations Later: A Bottom-up Course Design

Olle Willén
ollwi@ida.liu.se
Dept of Computer and Information Science
Linköpings universitet, Sweden
2006-02-20

ABSTRACT

A very traditional design of a basic course in programming may consist of repeated sequences of a lecture to introduce a few new concepts, followed by a lesson giving some practical examples, concluded with some hours of lab work on a compulsory task. When hours at disposal are limited you - the ambitious teacher - have to squeeze a lot of stuff into every lecture; one of them could for instance be assigned to the presentation of 'control structures'. Even if you don't have the ambition to be exhaustive you probably wish to generalize in order to show the wide range of alternatives, and as a practical consequence it is required that you also use some pieces of formal notation. Your intention is both to give a set of tools for students to choose from when they do their practical exercises, and at the same time to give an understanding of the generality of language elements. As a whole such a lecture could easily be considered as a bit abstract. In my experience many students will neither remember the higher level perspective (generalizations, formalisms) demonstrated in lectures of this kind, nor be quite able to extract the concrete instances they require for their tasks.

You could - perhaps easily - limit your teaching ambitions to just a few very specific cases, aiming at no more than what is really needed in the students' labs. At the university level this would however not be quite fair, neither to the students nor to your mission. A broader perspective must be maintained to satisfy students' varying needs, understanding, and degrees of curiosity, and to justify the level of a course as 'academic'.

The approach above demonstrates a kind of top-down teaching technique which could be appropriate in many learning situations. But with novices with no or little prior experience from formal notation or algorithmic thinking, and with unfamiliarity with the area's abstract ways of viewing, you should not take for granted that this will always work. The amount of information will often make students cognitively overloaded and confused. You certainly expect their area background to be very weak, and hence they have nothing to relate to, very little to build new knowledge from. The philosophy of constructivism emphasizes that new knowledge has to be built on already existing one, and you have to find out what that knowledge could be. Or, as an option, try to lay a foundation yourself.

The talk will present an alternative to the supposed traditional course design. The general idea is a bottom-up approach, where every introduction of new stuff begins with most practical computer based experiments and observations . These hands-on sessions are led and supervised by a teacher, and the obvious aim is to saw a seed of understanding. Not until later a lecture will overview the same concepts and from the experiences acquired earlier make the appropriate advancements and generalizations. However, since this design just recently has been adopted there is yet no evidence of what impact it may have.

Towards a Set of eXtreme Teaching Practices

Roy Andersson
Department of Computer Science
Lund Institute of Technology
Box 118
SE-221 00 Lund, Sweden
roy.andersson@cs.lth.se

Lars Bendix
Department of Computer Science
Lund Institute of Technology
Box 118
SE-221 00 Lund, Sweden
bendix@cs.lth.se

Abstract

Many universities have experienced an increase in the intake of students and at the same time cuts in the budgets for teaching. Many teachers have experienced that, for a number of reasons, they have to adjust their teaching or a course with short notice. These facts pose a challenge to the teachers' agility in adapting to changes – and in doing it in a cost efficient way.

We propose eXtreme Teaching (XT) as a framework that allows teachers to focus on experimenting with and improving their teaching techniques without compromising quality. The framework and the associated practices provide quick, accurate feedback that the teacher can act on. eXtreme Teaching will allow better student learning, better relationships with the students, increased interaction and development of the staff involved, less risk – and probably happier staff members.

In this paper, we describe the eXtreme Teaching framework and its nine practices.

Origin

The paper is presented and to be found in *Proceedings of the 5th International Conference on Computer Science Education, page 33-40*, Koli, Finland, November 17-20, 2005.

Conference web address:
http://www.cs.joensuu.fi/kolistelut/index.html

Proceedings web address:
http://www.it.utu.fi/koli05/proceedings/final_composition.b5.060207.pdf

Authentic Examination System

Torbjörn Jonsson
Department of Computer Science,
Linköpings Universitet, Sweden
torjo@ida.liu.se

ABSTRACT

Examination in a course can be done in several ways. The most common way is written examination where each student has a number of questions to answer. However, such examination is not similar to how the knowledge learned in the course will be use in real-life working condition. For a programming course a written examination is far from a real life working environment. We have therefore developed the Authentic Examination System (AES), which is a computer aided examination system. The aim of the system is to create an atmosphere more similar to the condition that a student will face when working as a programmer. We believe that important aspects are that in real-life condition, it is common to have your own assignment, it is possible to try (compile and run), you know when you are done, and colleagues may help a bit.

AES is a system where the student solution is corrected by the teacher in real-time during the examination. Note that AES is not an automatic examination system where the solutions are corrected automatically.

In a traditional written exam, each student is given a set of questions to answer. It is often so that the student has one or two occasions to ask the teacher clarifying questions regarding the exam. After the exam, the teacher corrects the solutions. In our AES the student has the opportunity to have constant communication during the examination.

In a traditional written examination the solution is corrected after the examination occasion and the result is sent to the student afterwards. It can be (up to) 2 weeks after the examination. However, in the AES, the solutions are received as they are produced by the students and we correct them immediately. The student has the result within a few minutes. Further, we also have the opportunity to give the student feedback on a solution during the exam. This gives the student opportunity to update the solution if it is not correct.

The examination is divided in two parts. The first part is examination of specific tasks (each task can be sent for correction separately). The second part is where we set the grade of the examination. The first part of the examination can be corrected by several teachers at the same time. This part gives one of the following results for a task:

Passed - The task is solved in a proper way and it is ok. This means that the student has completed the task.

Failed - The task is not correct and the student is not allowed to continue with it. It is still ok to send in others solutions on other tasks.

Not completed - The task is not good enough. The student may continue with the task and send in a new solution. If the student does not send in a new solution the result for the task is Failed.

The results are sent directly to the student during the examination, and at the same time the result is sent to the grading part (to the course examiner). Observe that the teachers can set results that the examiner may override. This leads to the fact that the students have this result as a preliminary result.

In the second part the examiner can override the result that the other teachers have set, but it is normally not the case. In this part the examiner sets the grade for the student. This grade is the "lowest" grade the student can get and this can be updated from a lower level to a higher if the student delivers more correct solutions in the examination.

The system allows the possibility to examine the student solutions in a "live" situation which leads to a learning situation even in the examination. The student also has the opportunity to decide when to stop. If the grade is at the maximum level the student can go home. Of course the student can go earlier if the result is good enough, however, we have noticed that most students stay as long as they are allowed to get the best grade as possible.

Of course the students and teachers have different views of the communication system. They also have different legal operations in the system. The student have personal information like name, person number, student identification "number", information about the course and time of examination. The student can (1) send questions, (2) receive answers, (3) send solutions, (4) receive results and comments on the solutions. The student has also information about the status of the examination like which task is done and what are the result of the examination of a task.

The teacher can (1) send information to all students or a specific one,(2) receive questions and solutions, (3) examine a student solution, (4) send results and comments of the solution and (5) much more.

An advantage with the system is that the students are anonymous to the teacher. That makes the examination better in the way that the teacher cannot use "old knowledge" learned during the course on a student such as "this student is smart". It also makes the examination form gender neutral.

Web-based UNIX training for first year students

Peter Dalenius
petda@ida.liu.se
Department of Computer and Information Science, Linköpings universitet
2006-02-20

The Department of Computer and Information Science hosts around 5000 student computer accounts with 1000 being first year students. The main working environment in the computer laboratories (75%) is UNIX, which is unknown to the majority of the new students. Many of our courses contain laboratory exercises, and the students spend a considerable amount of time using different pieces of software in our UNIX environment. It is important to us that the students are familiar with the operating system so that they can focus on the course contents instead of spending time on trivial tasks like moving files around or figuring out how to use the laser printer.

During the summer of 2000 I was given the task of developing a web-based course material for introducing new students to UNIX. I identified three main problems: a) The students come from different backgrounds and have different experiences. Some have experimented with UNIX-like systems before, many have some kind of basic ICT training, but the vast majority have never seen a UNIX system before. b) With the introductions used before, the students seemed to learn just enough to pass. They never saw the benefits of learning how to use the computer systems thoroughly. c) There was no clear overview of which ICT resources the department or the university offered to the students.

The course material was used during the fall of 2000, and during 2001 I developed the material further by designing and implementing a tool for presenting course material on the web called STONE (Simple Tool for Online Education). The aim was to create a tool that would be easy to use and extend, as opposed to some of the monolithic learning management systems on the market at that time. The STONE system offered three important features: a) The tool included self-correcting tests, mostly multiple choice, with questions chosen randomly from a set of alternatives. b) Each user could select the type of presentation he or she preferred. The course material had, at least to some extent, been written in two variants: one aimed at complete novices and one aimed at more experienced users. c) The separation of content and presentation made the material easy to update.

The STONE system and the course material have been used on all the major education programs in the department since 2001. The content has been updated several times to reflect changes in the computer systems as well as in student policies. In this paper and/or presentation I will describe my experiences from maintaining and using a web-based course material. I will present some results from student evaluations and discuss the course context in which the material has been used. Finally I will mention some ideas of how to develop the material and the use of it.

OMNotebook – Interactive WYSIWYG Book Software for Teaching Programming,

Anders Fernström, Ingemar Axelsson, Peter Fritzson, Anders Sandholm, Adrian Pop
PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden
{petfr,andsa,adrpo}@ida.liu.se

Abstract

OMNotebook is one of the first open source software systems that makes is possible to create interactive WYSIWYG books for teaching and learning programming. It has currently been used for course material (DrModelica) in teaching the Modelica language, but can easily be adapted to electronic books on teaching other programming languages, or even other subjects such as physics, chemistry, etc., where phenomena can be illustrated by dynamic simulations within the book. This could substantially improve teaching in a number of areas, including programming.

1 Need for more Interactive Learning

Traditional teaching methods are often too passive and do engage the student. A typical example is traditional lecturing.

Another typical learning method is reading a textbook on a subject matter. This is a good method, but sometimes requires a lot from the student. Also, learning programming needs interaction and programming exercises in order to grasp the concept.

A third way, would be to make the book active – be able to run programs and exercises within the book, and mix lecturing with doing exercises and reading in the interactive book.

2 Interactive Notebooks with Literate Programming

Interactive Electronic Notebooks are active documents that may contain technical computations and text, as well as graphics. Hence, these documents are suitable to be used for teaching and experimentation, simulation scripting, model documentation and storage, etc.

2.1 Mathematica Notebooks

Literate Programming (Knuth 1984) is a form of programming where programs are integrated with documentation in the same document. Mathematica notebooks (Wolfram 1997) is one of the first WYSIWYG (What-You-See-Is-What-You-Get) systems that support Literate Programming. Such notebooks are used, e.g., in the MathModelica modeling and simulation environment, e.g. see Figure 1 below and Chapter 19 in (Fritzson 2004)

2.2 OMNotebook

The OMNotebook software (Axelsson 2005, Fernström 2006) is a new open source free software that gives an interactive WYSIWYG (What-You-See-Is-What-You-Get) realization of Literate Programming, a form of programming where programs are integrated with documentation in the same document.

2.3 Tree Structured Hierarchical Document Representation

Traditional documents, e.g. books and reports, essentially always have a hierarchical structure. They are divided into sections, subsections, paragraphs, etc. Both the document itself and its sections usually have headings as labels for easier navigation. This kind of structure is also reflected in electronic notebooks. Every notebook corresponds to one document (one file) and contains a tree structure of cells. A cell can have different kinds of contents, and can even contain other cells. The notebook hierarchy of cells thus reflects the hierarchy of sections and subsections in a traditional document such as a book.

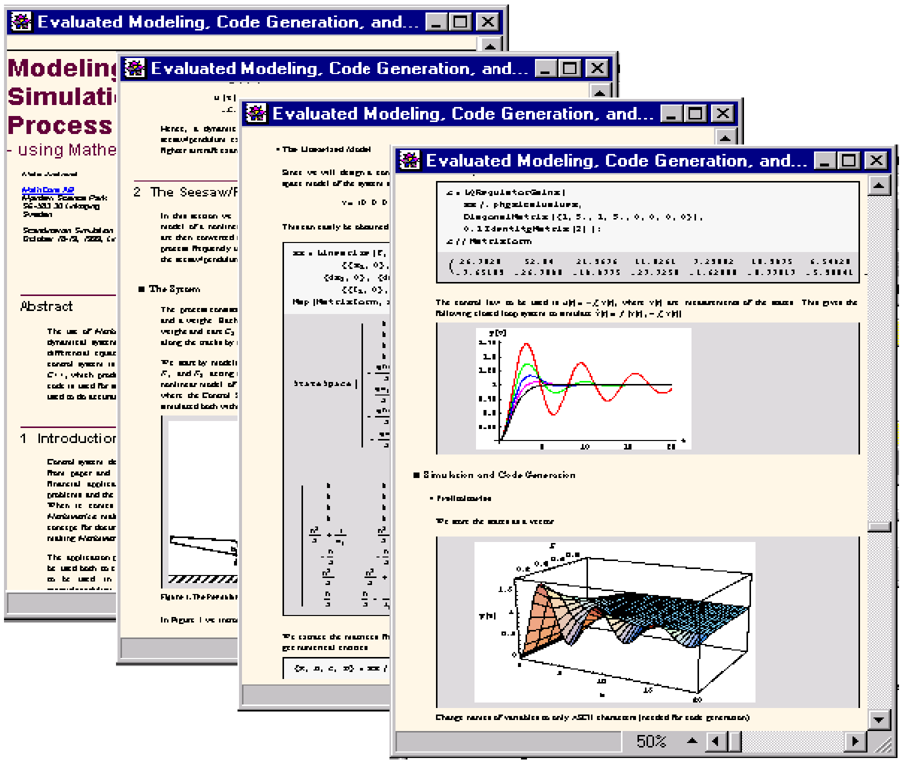

Evaluated Modeling, Code Generation, and...
Evaluated Modeling, Code Generation, and...
Evaluated Modeling, Code Generation, and...
Evaluated Modeling, Code Generation, and...
Modeling
Simulati
Process
- using Mathe
Abstract
1 Introduction
2 The Seesaw/P
The System
The Linearized Model
Simulation and Code Generation
50%

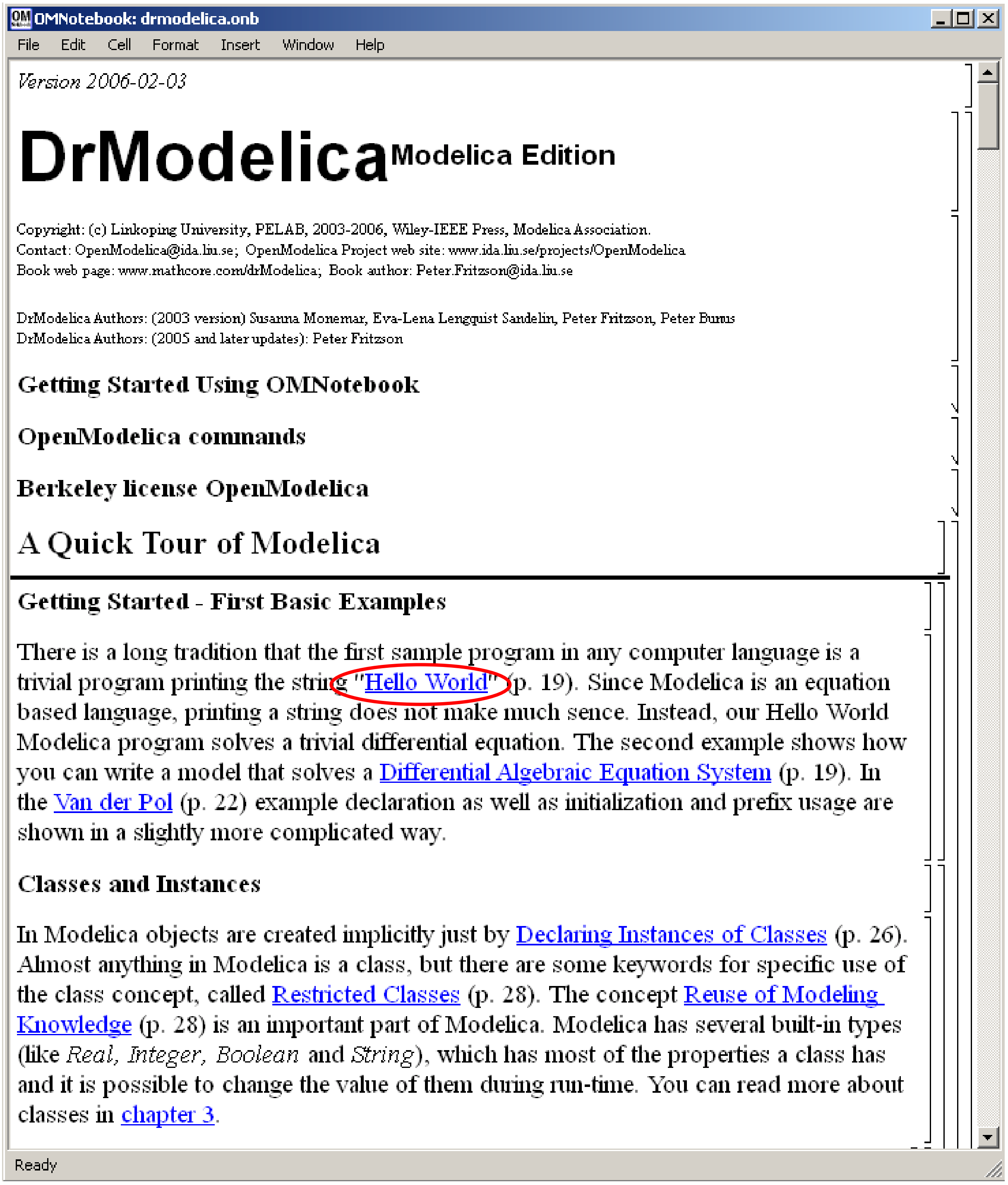
Figure 1. Examples of Mathematica notebooks in the MathModelica modeling and simulation environment.

3 The DrModelica Tutoring System – an Application of OMNotebook

Understanding programs is hard, especially code written by someone else. For educational purposes it is essential to be able to show the source code and to give an explanation of it at the same time.

Moreover, it is important to show the result of the source code's execution. In modeling and simulation it is also important to have the source code, the documentation about the source code, the execution results of the simulation model, and the documentation of the simulation results in the same document. The reason is that the problem solving process in computational simulation is an iterative process that often requires a modification of the original mathematical model and its software implementation after the interpretation and validation of the computed results corresponding to an initial model.

Most of the environments associated with equation-based modeling languages such as Modelica focus more on providing efficient numerical algorithms rather than giving attention to the aspects that should facilitate the learning and teaching of the language. There is a need for an environment facilitating the learning and understanding of Modelica. These are the reasons for developing the DrModelica teaching material for Modelica and for teaching modeling and simulation.


OMNotebook: drmodelica.onb
File Edit Cell Format Insert Window Help
Ready

Version 2006-02-03

DrModelica Modelica Edition

Copyright: (c) Linkoping University, PELAB, 2003-2006, Wiley-IEEE Press, Modelica Association.
Contact: OpenModelica@ida.liu.se; OpenModelica Project web site: www.ida.liu.se/projects/OpenModelica
Book web page: www.mathcore.com/drModelica; Book author: Peter.Fritzson@ida.liu.se

DrModelica Authors: (2003 version) Susanna Monemar, Eva-Lena Lengquist Sandelin, Peter Fritzson, Peter Bunus
DrModelica Authors: (2005 and later updates): Peter Fritzson

Getting Started Using OMNotebook

OpenModelica commands

Berkeley license OpenModelica

A Quick Tour of Modelica

Getting Started - First Basic Examples

There is a long tradition that the first sample program in any computer language is a trivial program printing the string "Hello World" (p. 19). Since Modelica is an equation based language, printing a string does not make much sence. Instead, our Hello World Modelica program solves a trivial differential equation. The second example shows how you can write a model that solves a Differential Algebraic Equation System (p. 19). In the Van der Pol (p. 22) example declaration as well as initialization and prefix usage are shown in a slightly more complicated way.

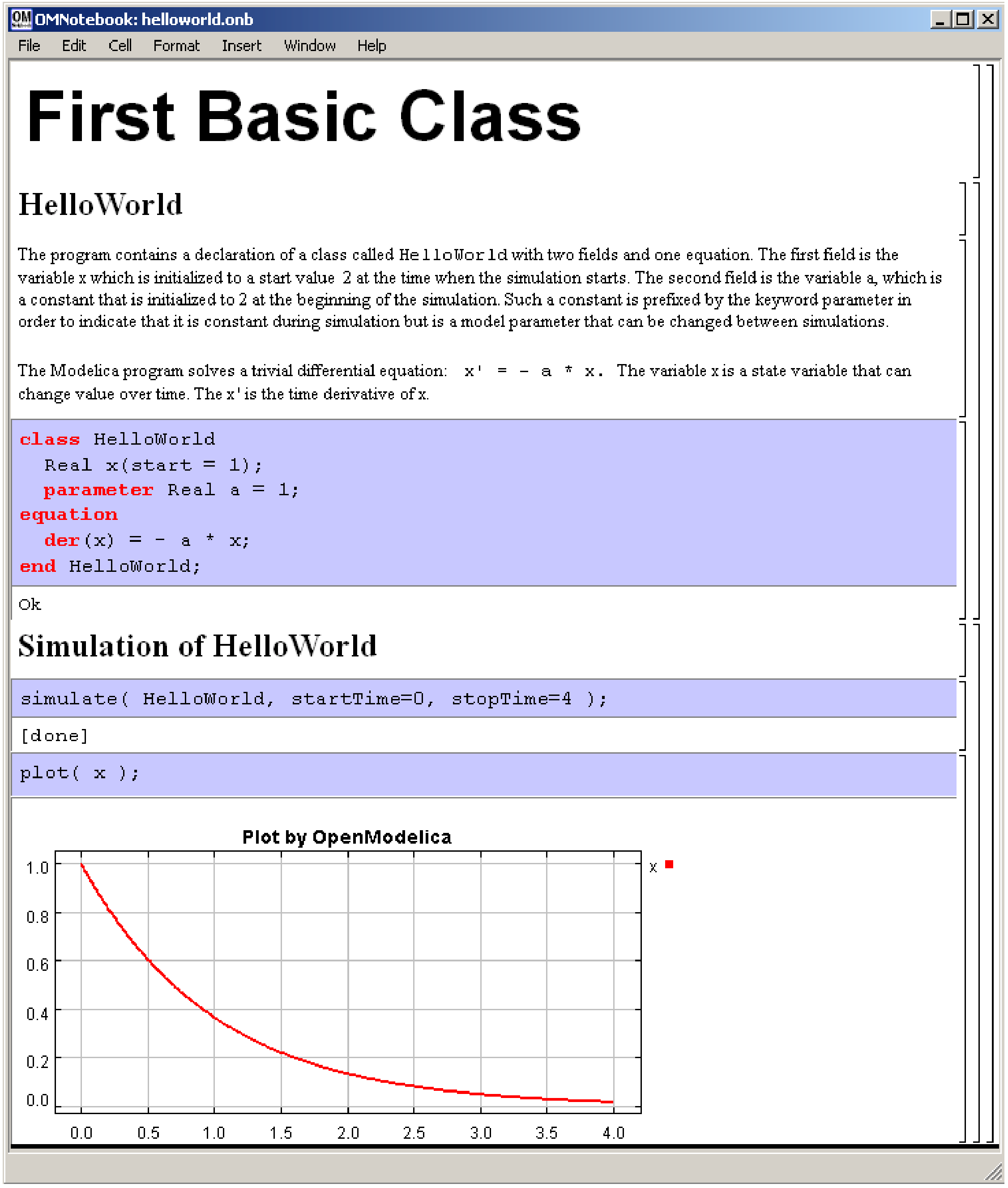
Classes and Instances

In Modelica objects are created implicitly just by Declaring Instances of Classes (p. 26). Almost anything in Modelica is a class, but there are some keywords for specific use of the class concept, called Restricted Classes (p. 28). The concept Reuse of Modeling Knowledge (p. 28) is an important part of Modelica. Modelica has several built-in types (like *Real, Integer, Boolean* and *String*), which has most of the properties a class has and it is possible to change the value of them during run-time. You can read more about classes in chapter 3.

Figure 2. The start page (main page) of the DrModelica tutoring system using OMNotebook. The link to the HelloWorld example shown in Figure 3 is marked with an oval.

DrModelica has a hierarchical structure represented as notebooks. The front-page notebook is similar to a table of contents that holds all other notebooks together by providing links to them. This particular notebook is the first page the user will see (Figure 2).

In each chapter of DrModelica the user is presented a short summary of the corresponding chapter of the book "Principles of Object-Oriented Modeling and Simulation with Modelica 2.1" by Peter Fritzson. The summary introduces some *keywords*, being hyperlinks that will lead the user to other notebooks describing the keywords in detail.


OMNotebook: helloworld.onb
File Edit Cell Format Insert Window Help
Plot by OpenModelica
x

First Basic Class

HelloWorld

The program contains a declaration of a class called HelloWorld with two fields and one equation. The first field is the variable x which is initialized to a start value 2 at the time when the simulation starts. The second field is the variable a, which is a constant that is initialized to 2 at the beginning of the simulation. Such a constant is prefixed by the keyword parameter in order to indicate that it is constant during simulation but is a model parameter that can be changed between simulations.

The Modelica program solves a trivial differential equation: x' = - a * x. The variable x is a state variable that can change value over time. The x' is the time derivative of x.

```
class HelloWorld
  Real x(start = 1);
  parameter Real a = 1;
equation
  der(x) = - a * x;
end HelloWorld;
```

```
Ok
```

Simulation of HelloWorld

```
simulate( HelloWorld, startTime=0, stopTime=4 );
```

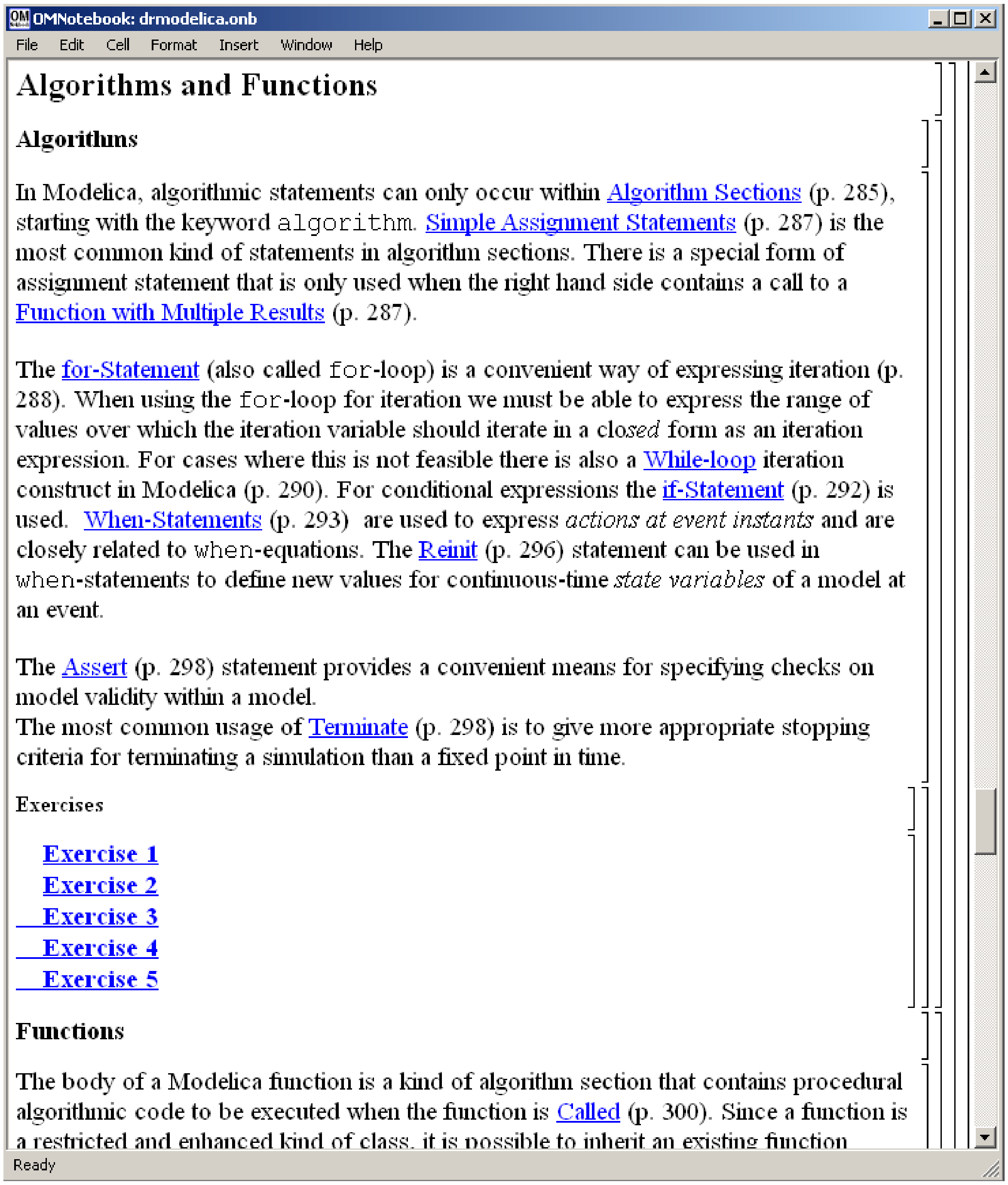
```
[done]
```

```
plot( x );
```

Figure 3. The `HelloWorld` class simulated and plotted using the OMNotebook version of DrModelica.

Now, let us consider that the link "*HelloWorld*" in DrModelica Section "Getting Started – First Basic Examples" in Figure 2 is clicked by the user. The new notebook, to which the user is being linked (see Figure 3), is not only a textual description but also contains one or more examples explaining the specific keyword. In this class, `HelloWorld`, a differential equation is specified.

No information in a notebook is fixed, which implies that the user can add, change, or remove anything in a notebook. Alternatively, the user can create an entirely new notebook in order to write his/her own programs or copy examples from other notebooks. This new notebook can be linked from existing notebooks.


OMNotebook: drmodelica.onb
File Edit Cell Format Insert Window Help
Ready

Algorithms and Functions

Algorithms

In Modelica, algorithmic statements can only occur within Algorithm Sections (p. 285), starting with the keyword `algorithm`. Simple Assignment Statements (p. 287) is the most common kind of statements in algorithm sections. There is a special form of assignment statement that is only used when the right hand side contains a call to a Function with Multiple Results (p. 287).

The for-Statement (also called `for`-loop) is a convenient way of expressing iteration (p. 288). When using the `for`-loop for iteration we must be able to express the range of values over which the iteration variable should iterate in a clo*sed* form as an iteration expression. For cases where this is not feasible there is also a While-loop iteration construct in Modelica (p. 290). For conditional expressions the if-Statement (p. 292) is used. When-Statements (p. 293) are used to express *actions at event instants* and are closely related to `when`-equations. The Reinit (p. 296) statement can be used in `when`-statements to define new values for continuous-time *state variables* of a model at an event.

The Assert (p. 298) statement provides a convenient means for specifying checks on model validity within a model.
The most common usage of Terminate (p. 298) is to give more appropriate stopping criteria for terminating a simulation than a fixed point in time.

Exercises

Exercise 1
Exercise 2
Exercise 3
Exercise 4
Exercise 5

Functions

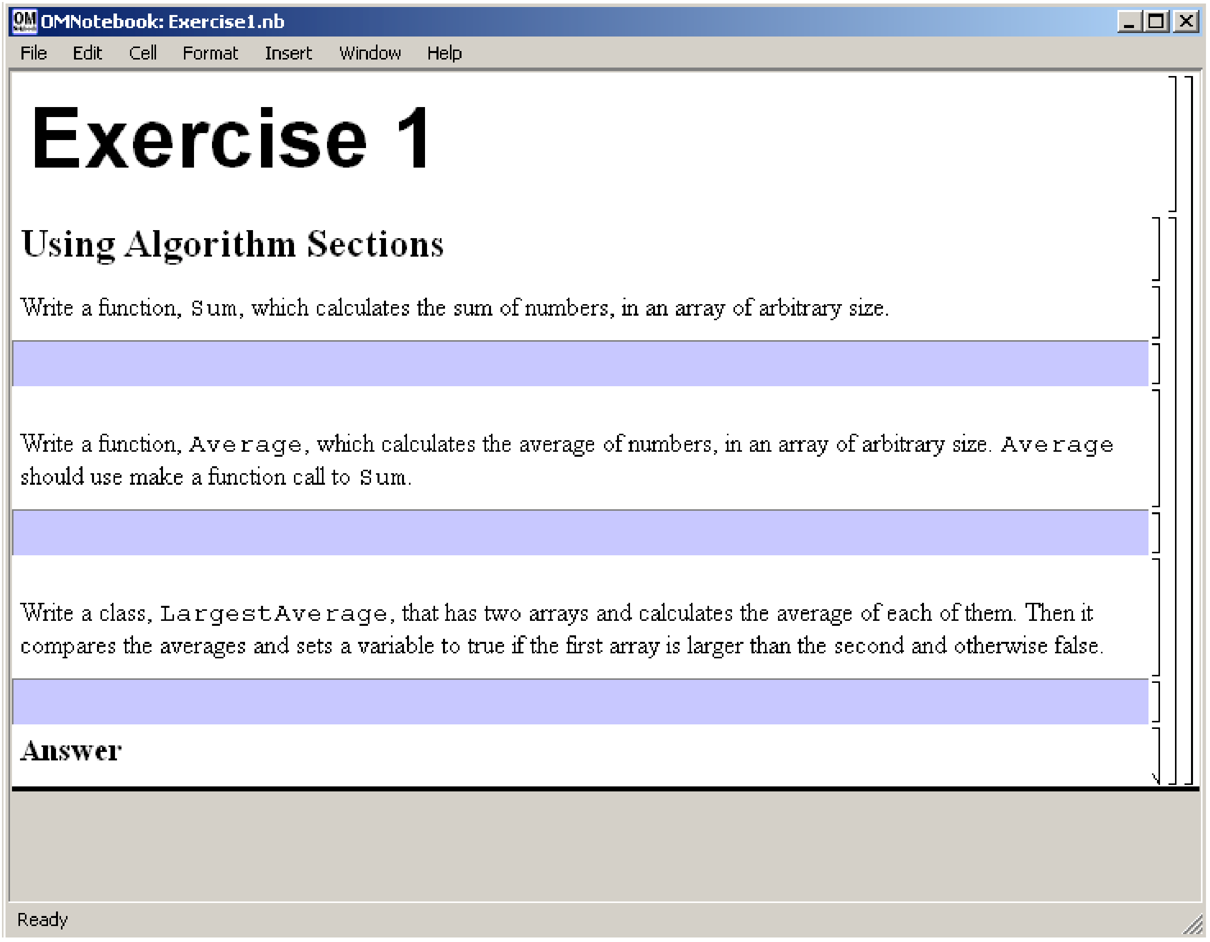
The body of a Modelica function is a kind of algorithm section that contains procedural algorithmic code to be executed when the function is Called (p. 300). Since a function is a restricted and enhanced kind of class, it is possible to inherit an existing function

Figure 4. DrModelica Chapter “Algorithms and Functions” in the main page of DrModelica.

When a class has been successfully evaluated the user can simulate and plot the result, as depicted in Figure 3 for the simple `HelloWorld` example model..

After reading a chapter in DrModelica the user can immediately practice the newly acquired information by doing the exercises that concern the specific chapter. Exercises have been written in order to elucidate language constructs step by step based on the pedagogical assumption that a student learns better "*using the strategy of learning by doing*". The exercises consist of either theoretical questions or practical programming assignments. All exercises provide answers in order to give the user immediate feedback.

Figure 4 shows the algorithm part of the Chapter "Algorithms and Functions" of the DrModelica teaching material. Here the user can read about Modelica language constructs, like `algorithm` sections, when-statements, and `reinit` equations, and then practice these constructs by solving the exercises corresponding to the recently studied section.



Exercise 1

Using Algorithm Sections

Write a function, `Sum`, which calculates the sum of numbers, in an array of arbitrary size.

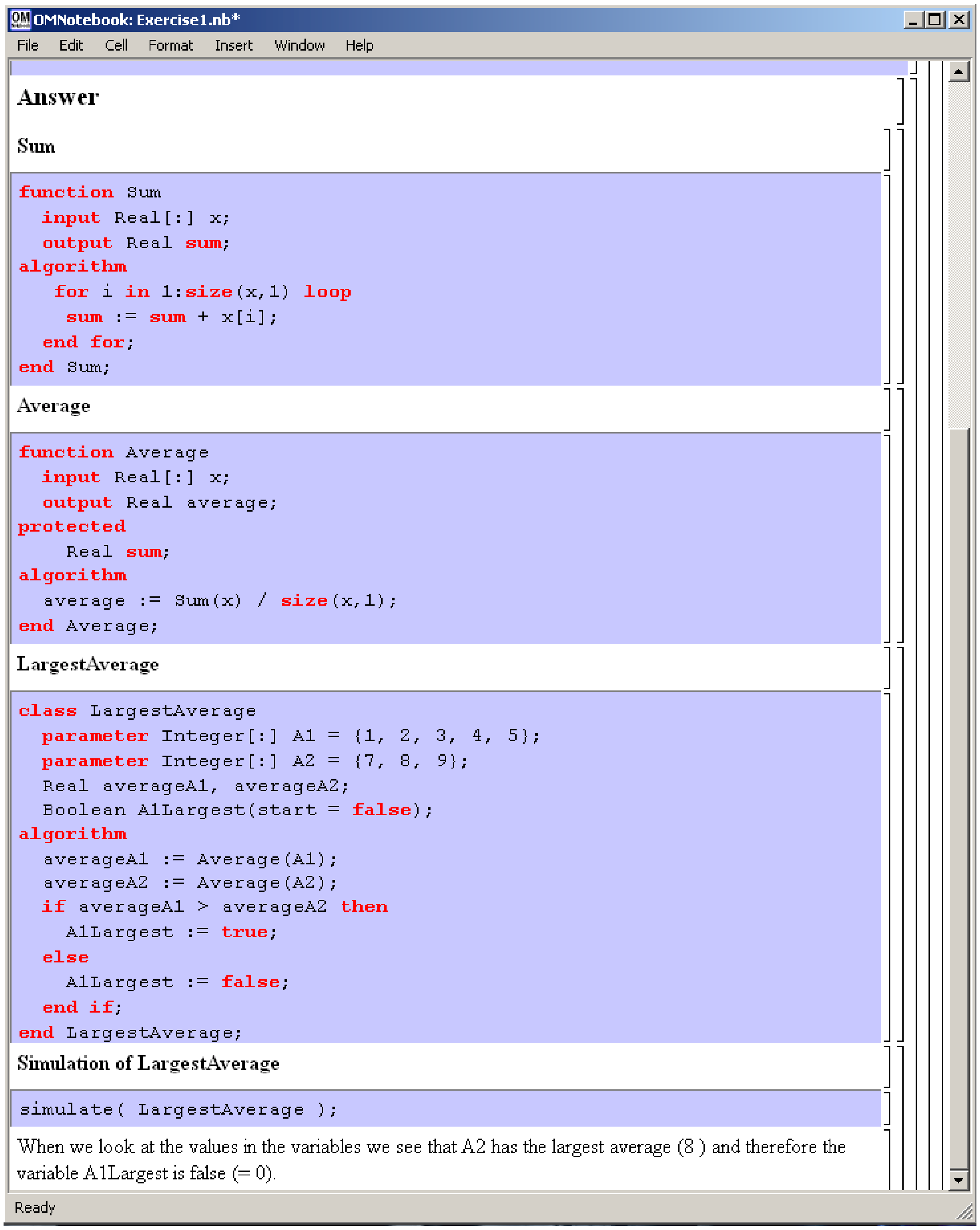
Write a function, `Average`, which calculates the average of numbers, in an array of arbitrary size. `Average` should use make a function call to `Sum`.

Write a class, `LargestAverage`, that has two arrays and calculates the average of each of them. Then it compares the averages and sets a variable to true if the first array is larger than the second and otherwise false.

Answer

Figure 5. Exercise 1 in Chapter "Algorithms and Functions" of DrModelica.

Exercise 1 in the algorithm part of Chapter "Algorithms and Functions" is shown in Figure 5. In this exercise the user has the opportunity to practice different language constructs and then compare the solution to the answer for the exercise. Notice that the answer is not visible until the *Answer* section is expanded. The answer is shown in Figure 6.


OMNotebook: Exercise1.nb*
File Edit Cell Format Insert Window Help

Answer

Sum

```
function Sum
  input Real[:] x;
  output Real sum;
algorithm
   for i in 1:size(x,1) loop
    sum := sum + x[i];
  end for;
end Sum;
```

Average

```
function Average
  input Real[:] x;
  output Real average;
protected
    Real sum;
algorithm
  average := Sum(x) / size(x,1);
end Average;
```

LargestAverage

```
class LargestAverage
  parameter Integer[:] A1 = {1, 2, 3, 4, 5};
  parameter Integer[:] A2 = {7, 8, 9};
  Real averageA1, averageA2;
  Boolean A1Largest(start = false);
algorithm
  averageA1 := Average(A1);
  averageA2 := Average(A2);
  if averageA1 > averageA2 then
    A1Largest := true;
  else
    A1Largest := false;
  end if;
end LargestAverage;
```

Simulation of LargestAverage

```
simulate( LargestAverage );
```

When we look at the values in the variables we see that A2 has the largest average (8) and therefore the variable A1Largest is false (= 0).

Figure 6. The answer section to Exercise 1 in Chapter “Algorithms and Functions” of DrModelica.

4 Conclusions

The OMNotebook software is one of the first open source software systems that makes is possible to create interactive WYSIWYG books for teaching and learning programming. It has currently been used for course material (DrModelica) in teaching the Modelica language, but can easily be adapted to electronic books on teaching other programming languages such as Java, Scheme, etc, through its general CORBA interface. This could revolutionize teaching in programming.

5 Acknowledgements

Support for Modelica-related research from SSF and Vinnova is gratefully acknowledged. Eva-Lena Lengquist-Sandelin and Susanna Monemar prepared the first version of the DrModelica tutorial material and thus also contributed to this paper. Daniel Hedberg at MathCore provided advice regarding Qt-based implementation. Peter Aronsson implemented large parts of the OpenModelica compiler and the communication protocol used from OMNotebook. PhD students at PELAB (Programming Environment Lab) contributed to various aspects of OpenModelica.

References

[1] Eric Allen, Robert Cartwright, Brian Stoler. DrJava: A lightweight pedagogic environment for Java. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education* (SIGCSE 2002) (Northern Kentucky – The Southern Side of Cincinnati, USA, February 27 – March 3, 2002).

[2] Ingemar Axelsson. *OpenModelica Notebook for Interactive Structured Modelica Documents*. Final thesis, LITH-IDA-EX–05/080–SE, Linköping University, Linköping, Sweden, October 21, 2005.

[3] Anders Fernström. *Extending OMNotebook – An Interactive Notebook for Structured Modelica Documents.*Final thesis to be presented spring 2006, Dept. Computer and Information Science, Linköping University, Sweden.

[4] Peter Fritzson. *Principles of Object Oriented Modeling and Simulation with Modelica 2.1*, 940 pages, ISBN 0-471-471631, Wiley-IEEE Press. Feb. 2004.

[5] Peter Fritzson, et al. *OpenModelica Users Guide*, Preliminary Draft, for OpenModelica 1.3.1, Nov 28 2005. www.ida.liu.se/projects/OpenModelica.

[6] Knuth, Donald E. Literate Programming. *The Computer Journal*, NO27(2), pp. 97–111, May 1984.

[7] Eva-Lena Lengquist-Sandelin, Susanna Monemar, Peter Fritzson, and Peter Bunus. DrModelica – A Web-Based Teaching Environment for Modelica. In *Proceedings of the 44th Scandinavian Conference on Simulation and Modeling* (SIMS'2003), available at www.scan-sims.org. Västerås, Sweden. September 18-19, 2003.

[8] The Modelica Association. The Modelica Language Specification Version 2.2, March 2005. http://www.modelica.org.

[9] Stephen Wolfram. *The Mathematica Book*. Wolfram Media Inc, 1997.

Dealing with tasks in a realistic object-oriented system

- The first result: understandings of the interface concept in Java

Jonas Boustedt
Department of Information Technology
Uppsala University,
SE-75105 Uppsala, Sweden
Computer Science Department
University of Gävle
SE-80176 Gävle, Sweden
jbt@hig.se
30 November 2005

ABSTRACT

Courses in object-oriented programming cover abstract concepts which in many cases make sense only in larger scale software. However, our examples are often narrowed down to fit the traditional way of teaching and in the practical assignments the students are expected to build their solutions from scratch. We are curious about variations in how students understand the concepts in a wider, more realistic context and variations in their approaches. This curiosity origins from our interest in teaching computer science and from our ambition to promote the students' professional careers. This is an empirical study with a phenomenographic approach where data is collected using a video recorded role play and a following interview. The students acted as "rookies" who were supposed to complete a software system, developed by a senior programmer. The result of the analysis, so far, is a categorisation of qualitatively different understandings of the Java interface, on a collective level. However, more results are expected from this particular study, such as understandings of other concepts, experiences of the software system and different approaches to solve the task.

Teaching parallel programming early

Christoph W. Kessler
Institute for Computer and Information Science (IDA)
Linköping university, S-58183 Linköping, Sweden
chrke@ida.liu.se

Abstract— **In this position paper, we point out the importance of teaching a basic understanding of parallel computations and parallel programming early in computer science education, in order to give students the necessary expertise to cope with future computer architectures that will exhibit an explicitly parallel programming model.**

We elaborate on a programming model, namely shared-memory bulk-synchronous parallel programming with support for nested parallelism, as it is both flexible (can be mapped to many different parallel architectures) and simple (offers a shared address space, structured parallelism, deterministic computation, and is deadlock-free).

We also suggest taking up parallel algorithmic paradigms such as parallel divide-and-conquer together with their sequential counterparts in the standard CS course on data structures and algorithms, in order to anchor thinking in terms of parallel data and control structures early in the students' learning process.

I. Introduction

For 50 years we have been teaching students programming, algorithms and data structures with a programming model dominated by the sequential von-Neumann architecture. This model is popular because of its simplicity of control flow and memory consistency and its resemblance to the functionality of early computer architectures.

However, recent trends in computer architecture show that the performance potential of von-Neumann programming has finally reached its limits. Computer architectures even for the desktop computing domain are, at least under the hood, increasingly parallel, e.g. in the form of multithreaded processors and multi-core architectures. The efforts for grafting the von-Neumann model on top of thread-level parallel and instruction-level parallel processor hardware, using techniques such as dynamic instruction dispatch, branch prediction or speculative execution, are hitting their limits, in the form of high design complexity, limited exploitable instruction-level parallelism in applications, and power and heat management problems. We foresee that explicitly parallel computing models will emerge in the near future to directly address the massive parallelism available in upcoming processor architectures. In order to fully exploit their performance potential, applications will have to be parallelized, that is, be (re)written to exhibit explicit parallelism. However, most programmers are reluctant to adopting a parallel programming model, (1) because parallel programming is notoriously more complex and error-prone than sequential programming, at least with the parallel programming systems that are in use today, and (2) because most programmers were never trained in *thinking* in terms of parallel algorithms and data structures.

For the years to come, explicitly parallel programming paradigms will have to be adopted by more and more programmers. In this position paper, we suggest preparing students in time for this paradigm shift. A first step could be to take up parallel computing issues relatively early in existing standard courses, e.g. in the undergraduate course on data structures and algorithms. Many fundamental algorithmic concepts such as divide-and-conquer have an immediate parallel counterpart, which may be considered together. The goal is to anchor thinking in parallel structures early in the education process.

In particular, we advocate *simple* parallel programming models, such as the *bulk-synchronous parallel* (BSP) model, because they are (a) still flexible enough to be mapped to a wide range of parallel architectures, and (b) simple enough to provide a good basis for the design and analysis of parallel algorithms and data structures that offers a compatible extension of the existing theory.

The remainder of this paper is organized as follows. In Section II we summarize the most important parallel programming models and discuss their suitability as a platform for teaching parallel programming early. Section III elaborates on one particular parallel programming model and language, NestStep. We discuss more teaching issues in Section IV, and Section V concludes.

II. Survey of parallel programming models

With the need to exploit explicit parallelism at the application programming level, programmers will have

TABLE I
SURVEY OF PARALLEL PROGRAMMING MODELS

Progr. Model	Control Structure	Data View	Consistency	Main Restriction	Examples
Message Passing	Asynchronous, MIMD	Local	N.a.	None	MPI
Shared Memory	Asynchronous, MIMD	Shared	Weak	None	Pthreads, OpenMP, UPC, Cilk
Data-Parallel	Synchronous, SIMD	Shared	Strict	SIMD-like Control	HPF, C^*
PRAM	Synchronous, MIMD	Shared	Strict	None	Fork
BSP	Bulk-synchr., MIMD	Local	N.a.	Superstep Structure	BSPlib, PUB
NestStep-BSP	Nested BSP	Shared	Superstep	Superstep Structure	NestStep

to adopt a parallel programming model. In this section, we briefly review the most important ones.

A. *Message passing*

The currently predominant model for programming supercomputers is message passing with MPI, the message-passing interface [13], which is a portable but low-level standard for interprocessor communication. Message passing may be considered a least common denominator of parallel computing, which is available on almost every parallel computer system that is in use today. Message passing code is generally unstructured and hard to understand, maintain and debug. It only supports a local address space (starting from location 0 on each processor) and requires the programmer to place explicit send and receive operations in the code, and maybe even handle buffering explicitly, in order to exchange data between processors. Modest improvements such as one-sided communication (automatic receive), nested parallelism (by processor group splitting), and collective communication operations (such as reductions, scatter and gather) help to somewhat reduce complexity—usually at the expense of some performance loss—but are not enforced, such that unstructured parallelism is still the default.

B. *Shared memory and shared address space*

Shared-memory parallel computing is, in practice, often realized by asynchronously operating threads or processes that share memory. Synchronization between these parallel activities (here referred to as processors, for simplicity) can be in the form of asynchronous signals or mutual exclusion in various forms (semaphores, locks, monitors etc.). A global address space provides to the programmer a more natural interface to access data that is compliant with sequential computing models. Unstructured shared-memory parallel programming platforms such as *pthreads* have been complemented by more structured languages such as *OpenMP* [12], which supports work-sharing constructs to schedule parallel tasks such as parallel loop iterations etc. onto a fixed set of processors. Additionally, shared-address-space languages such as UPC [2] that emulate a shared memory view on top of a message passing platform have emerged recently. On the other hand, memory consistency must increasingly be handled explicitly by the programmer at a fairly low level, e.g. by *flush* operations that reconcile the local value of a cached memory location with its main memory value. Finally, none of these platforms really supports nested parallelism.

C. *Data-parallel computing*

Data-parallel computing is the software equivalent of SIMD (single instruction stream, multiple data streams) architectures. Processors share a single program control, that is, execute at the same time the same operation on maybe different data (or do nothing). Data-parallel computing was very popular in the 1980's and early 1990's because it mapped directly to the vector processors and array computers of that period. Many data-parallel programming languages have been developed, most notably High-Performance Fortran (HPF) [4]. Generally, data-parallel languages offer a shared address space with a global view of large data structures such as matrices, vectors etc. Data-parallel computing is suitable for regular computations on large arrays but suffers from inflexible control and synchronization structure in irregular applications, for which a MIMD (multiple instruction streams, multiple data streams) based model is more appropriate. Although HPF is not widely used in practice, many of its concepts have found their way into standard Fortran and into certain shared-address-space languages.

D. *PRAM model*

In the design and analysis of parallel algorithms, we mainly work with three theoretical models that are all extensions of the sequential RAM model (Random Access Machine, also known as the von-Neumann model): the PRAM, the BSP, and systolic arrays.

The PRAM (Parallel Random Access Machine) model, see the book by Keller, Kessler and Träff [6] for an introduction, connects a set of P processors to a single shared memory module and a common clock signal. In a very idealistic simplification, shared memory access time is assumed to be uniform and take one clock cycle, exactly as long as arithmetic and branch operations. Hence, the entire machine operates synchronously at the instruction level. If simultaneous accesses to the same memory location by multiple processors are resolved in a deterministic way, the resulting parallel computation will be deterministic. Memory will always be consistent. Synchronization can be done via signals, barriers, or mutual exclusion; deadlocks are possible. As the shared memory becomes a performance bottleneck without special architectural support, the PRAM model has not been realized in hardware, with one notable exception, the SB-PRAM research prototype at Saarbrücken university in the 1990s [6].

E. BSP model

In contrast, the BSP (Bulk-Synchronous Parallel) model [15] is an abstraction of a restricted message passing architecture and charges a cost for communication. The machine is characterized by only three parameters: the number of processors P, the byte transfer rate g in point-to-point communication, and the overhead L for a global barrier synchronization (where g and L may be functions of P). The BSP programmer must organize his/her parallel computation as a sequence of *supersteps* that are conceptually separated by global barriers. Hence, the cost of a BSP computation is the sum over the cost of every superstep. A superstep (see also Figure 1) consists of a *computation phase*, where processors only can access local memory, and a subsequent *communication phase*, where processors exchange values by point-to-point message passing. Hence, BSP programs have to use local addresses for data, and the programmer needs to write explicit send and receive statements for communication. Routines for communication and barrier synchronization are provided in BSP libraries such as BSPlib [5] and PUB [1]. Nested parallelism is not supported in classical BSP; in order to exploit nested parallelism in programs, it must be flattened explicitly by the programmer, which is generally difficult for MIMD computations. Hence, although announced as a "bridging" model (*i.e.*, more realistic than PRAM but simpler to program and analyze than completely unstructured message passing), some of the problems of message passing programming are inherited. We will show in the next section how to relax these constraints.

F. Which model is most suitable?

Table I summarizes the described parallel programming models, with their main strenghts and weaknesses.

From a technical point of view, a model that allows to control the underlying hardware architecture efficiently is most important in high-performance computing. This explains the popularity of low-level models such as MPI and pthreads.

From the educational point of view, a simple, deterministic, shared memory model should be taught as a first parallel programming model. While we would opt for the PRAM as model of choice for this purpose [10], we are aware that its overabstraction from existing parallel computer systems may cause motivation problems for many students. We therefore consider BSP as a good compromise, as it is simple, deterministic, semi-structured and relatively easy to use as a basis for quantitative analysis, while it can be implemented on a wide range of parallel platforms with decent efficiency. However, to make it accessible to masses of programmers, it needs to equipped with a shared address space and better support for structured parallelism, which we will elaborate on in the next section.

The issues of data locality, memory consistency, and performance tuning remain to be relevant for high-performance computing in practice, hence interested students should also be exposed to more complex parallel programming models at a later stage of education, once the fundamentals of parallel computing are well understood.

III. NestStep

NestStep [8], [7] is a parallel programming language based on the BSP model. It is defined as a set of language extensions that may be added, with minor modifications, to any imperative programming language, be it procedural or object oriented. The sequential aspect of computation is inherited from the basis language. The new NestStep language constructs provide shared variables and process coordination. The basis language need not be object oriented, as parallelism is not implicit by distributed objects communicating via remote method invocation, but expressed by separate language constructs.

NestStep processes run, in general, on different machines that are coupled by the NestStep language extensions and runtime system to a virtual parallel computer. Each processor executes one process with the same program (SPMD), and the number of processes remains constant throughout program execution.

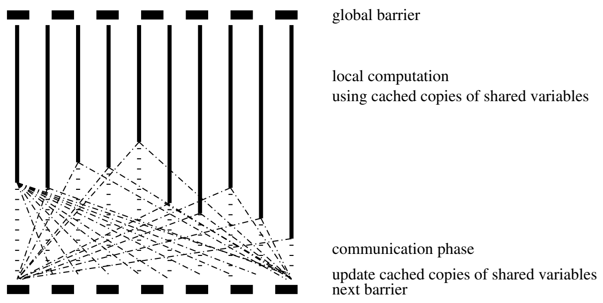

global barrier
local computation
using cached copies of shared variables
communication phase
update cached copies of shared variables
next barrier

Fig. 1. A BSP superstep. — In NestStep, supersteps form the units of synchronization and shared memory consistency.

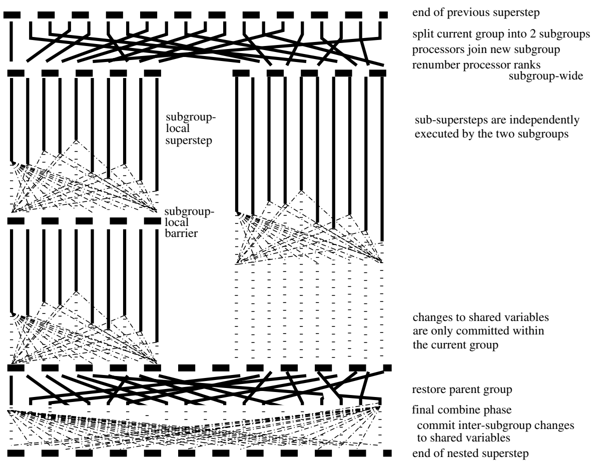

end of previous superstep
split current group into 2 subgroups
processors join new subgroup
renumber processor ranks
subgroup-wide
subgroup-local superstep
sub-supersteps are independently executed by the two subgroups
subgroup-local barrier
changes to shared variables are only committed within the current group
restore parent group
final combine phase
commit inter-subgroup changes to shared variables
end of nested superstep

Fig. 2. Nesting of supersteps, here visualized for a `neststep(2,...) ...` statement splitting the current group into two subgroups. Dashed horizontal lines represent implicit barriers.

The NestStep processors are organized in *groups*. The processors in a group are automatically ranked from 0 to the group size minus one. The `main` method of a NestStep program is executed by the *root group* containing all available processors of the partition of the parallel computer the program is running on. A processor group executes a BSP superstep as a whole. Ordinary, flat supersteps are denoted in NestStep by the `step` statement

`step` *statement*

Groups can be dynamically subdivided during program execution, following the static nesting structure of the supersteps. The current processor group is split by the `neststep` statement, as in

```
neststep(2; @=(cond)?1:0) // split group
  if (@==1) stmt1();
  else      stmt2();
```

into several subgroups, which can execute supersteps (with local communication and barriers) independent of each other. At the end of the neststep statement, the subgroups are merged again, and the parent group is restored. See Figure 2 for an illustration. Note that this corresponds to forking an explicitly parallel process into two parallel subprocesses, and joining them again.

Group splitting can be used immediately for expressing parallel divide-and-conquer algorithms.

Variables in NestStep are declared to be either *shared* (`sh`) or *private*. A private variable exists once on each processor and is only accessible locally. A shared variable, such as `sum` in Figure 3, is generally replicated: one copy of it exists on each processor. The NestStep runtime system guarantees the *superstep consistency* invariant, which says that at entry and exit of a superstep, the values of all copies of a replicated shared variable will be equal. Of course, a processor may change the value of its local copy in the computation phase of a superstep. Then, the runtime system will take special action to automatically make all copies of that variable consistent again, during the communication phase of the superstep. The conflict resolution strategy for such concurrent writes can be programmed individually for each variable (and even for each superstep, using the `combine` clause). For instance, we could specify that an arbitrary updated value will be broadcast and committed to all copies, or that a reduction such as the global sum of all written values will be broadcast and committed. As the runtime system uses a communication tree to implement this combining of values, parallel reduction and even prefix computations can be performed on-the-fly without additional overhead [8]. Exploiting this feature, parallel prefix computations, which are a basic building block of many parallel algorithms, can be written in a very compact and readable way, see Figure 3.

Shared arrays can be either replicated as a whole, or distributed (each processor owns an equally large part of it), as `a` in the example program in Figure 3. Distributed shared arrays are complemented by appropriate iterator constructs. For instance, the forall loop

```
forall ( i, a )
   stmt(i, a[i]);
```

scans over the entire array `a` and assigns to the private iteration counter `i` of each processor exactly those indices of `a` that are locally stored on this processor.

As superstep computations only allow access to locally available elements, values of remote elements needed by a processor must be fetched before entry to a superstep, and written elements will be shipped (and combined) in the communication phase at the end of a superstep. Fetching array elements beforehand can be

```
void parprefix( sh int a[]</> )
{
 int *pre;          // priv. prefix array
 int p=#, Ndp=N/p; // assume p divides N
 int myoffset;      // my prefix offset
 sh int sum = 0;
 int i, j = 0;
 step {
   pre = new_Array( Ndp, Type_int );
   forall ( i, a ) {  // owned elements
     pre[j++] = sum;
     sum += a[i];
   }
 } combine( sum<+:myoffset> );
 j = 0;
 step
   forall ( i, a )
     a[i] = pre[j++] + myoffset;
}
```

Fig. 3. Computing parallel prefix sums in NestStep-C.

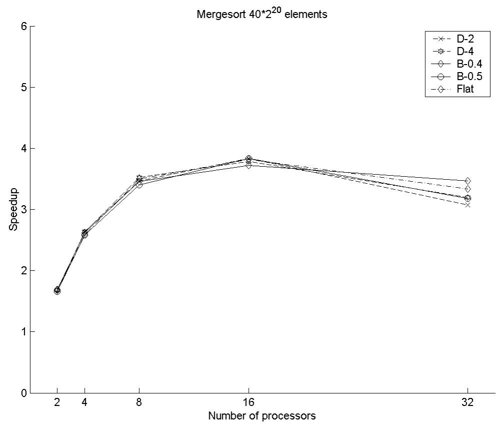


Fig. 4. Relative speedup of a parallel mergesort implementation in NestStep where the merge function is not parallelized. Measurements were done on a Linux cluster; the different speedup curves correspond to different configurations of the NestStep runtime system [14].

a problem in irregular computations where the actual elements to be accessed are not statically known. A technique for scheduling the necessary two-sided communication operations on top of NestStep's combining mechanism for replicated variables is described in [7].

NestStep does not support mutual exclusion, and is thus deadlock-free. The main synchronization primitive is the barrier synchronization included in the `step` and `neststep` statements. In many cases, the need for mutual exclusion disappears as it can be expressed by suitably programming the concurrent write conflict resolution of shared variables. Otherwise, the design pattern for serializing computation is to determine the next processor to do the critical computation, usually by a prefix combine operation at the end of a superstep, and then masking out all but that processor in the computation phase of the following superstep.

At the time of writing, the run-time system of NestStep, implemented on top of MPI, is operational. In measurements on a Linux cluster supercomputer, NestStep outperformed OpenMP (running on top of a distributed-shared memory emulation layer) by a factor of up to 30 [14]. A front end (compiler) for NestStep is in preparation.

IV. Teaching parallel programming

A. *Parallel algorithmic paradigms*

Many modern textbooks about (sequential) algorithms teach algorithmic concepts and then present one or several algorithms as incarnation of that concept. Many of these concepts indeed have also a direct parallel counterpart.

For instance, *divide-and-conquer* (DC) is an important algorithmic problem solving strategy in sequential computing. A problem is split into one or more independent subproblems of smaller size (divide phase); each subproblem is solved recursively (or directly if it is trivial) (conquer phase), and finally the subsolutions are combined to a solution of the original problem instance (combine phase). Examples for DC computations are mergesort, FFT, Strassen matrix multiplication, Quicksort or Quickhull.

In parallel computing, the *parallel DC* strategy allows for a simultaneous solution of all subproblems, because the subproblems are independent of each other. Parallel DC requires language and system support for nested parallelism, because new parallel activities are created in each recursion step. However, a significant speedup can generally be obtained only if also the work-intensive parts of the divide and the combine phase can be parallelized. This effect is often underestimated by students. For instance, Figure 4 shows the relative speed-up curve for a student's implementation of parallel mergesort that exploits the independence of subproblems in the DC structure of mergesort but uses a sequential merge function for combining the sorted subarrays, which leads to parallel time $O(n)$ with n processors and elements, rather than $O(\log^2 n)$ which can be achieved if the merge function is parallelized as well to run in logarithmic parallel time. If this phenomenon is not made transparent to students in the form of a *work-time-cost analysis framework* (see e.g. [6]), they wonder why their "fully parallel" code does not scale to large numbers of processors but shows saturation effects, as in Figure 4.

B. Textbooks and curricular issues

At Linköping university, about 300 undergraduate students in computer science and closely related study programs per year take one of our fundamental courses on data structures and algorithms, usually in the second year. However, only few of these (ca. 40 per year) find their way into the optional, final-year course on parallel programming, and only 4–8 PhD students every second year sign up for a PhD-level course in parallel programming. Beyond unavoidable diversification and specialization in the late study phases, this may also be caused by a lack of anchoring a fundamental understanding of parallel computing in undergraduate education and, as a consequence, in the mind of the students.

Part of this effect may be attributed to an underrepresentation of parallel computing issues in the established course literature on algorithms. Standard textbooks on algorithms generally focus on sequential algorithms. Some also contain a chapter or two on parallel algorithms, such as Cormen et al. [3]. Up to now, we only know of a single attempt to provide a unified treatment of both sequential and parallel algorithms in a single textbook, by Miller and Boxer [11]. However, their text, albeit fairly short, covers many different parallel programming models and interconnection network topologies, making the topic unnecessarily complex for a second-year student.

Instead, we propose to take up parallel algorithmic paradigms and some example parallel algorithms in a second-year algorithms course, but based on a *simple parallel programming model*, for instance an enhanced BSP model as supported by NestStep.

C. Experiences in special courses on parallel computing

In a graduate-level course on parallel programming models, languages and algorithms, we used the PRAM model and the C-based PRAM programming language Fork [6], which is syntactically a predecessor of NestStep. That course contained a small programming project realizing a bitonic-sort algorithm as presented in theory in Cormen et al. [3] in Fork and running and evaluating it on the SBPRAM simulator [9]. The goal was that the students should understand the structure of the algorithm, analyze and experimentally verify the time complexity ($O(\log^2 N)$ with N processors to sort N elements). Our experiences [10] indicate that the complementation of the theoretical description of the parallel algorithm by experimental work was appreciated, and that the available tools (such as a trace file visualizer showing the computation and group structure) helped in developing and debugging the program. We expect similar results for NestStep once a frontend will be available.

V. Conclusion

We have motivated why technical and teaching support for explicit parallel programming gets more and more important in the coming years. We have reviewed the most important parallel programming models, elaborated on an enhanced version of the BSP model, and described the NestStep parallel programming language supporting that model. Overall, we advocate a simple, structured parallel programming model with deterministic consistency and synchronization mechanism, which should be accessible to a larger number of students and presented earlier in the curriculum to create a fundamental understanding of parallel control and data structures. We suggested a scenario how parallel programming concepts could be added into a standard course on algorithms, that is, relatively early in computer science education.

References

[1] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) Library. *Parallel Computing*, 29:187–207, 2003.

[2] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, second printing, IDA Center for Computing Sciences, May 1999.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[4] High Performance Fortran Forum HPFF. High Performance Fortran Language Specification. *Sci. Progr.*, 2, 1993.

[5] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: the BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, 1998.

[6] Jörg Keller, Christoph Kessler, and Jesper Träff. *Practical PRAM Programming*. Wiley, New York, 2000.

[7] Christoph Kessler. Managing distributed shared arrays in a bulk-synchronous parallel environment. *Concurrency – Pract. Exp.*, 16:133–153, 2004.

[8] Christoph W. Keßler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The J. of Supercomputing*, 17:245–262, 2000.

[9] Christoph W. Keßler. Fork homepage, with compiler, SBPRAM simulator, system software, tools, and documentation. www.ida.liu.se/∼chrke/fork/, 2001.

[10] Christoph W. Kessler. A practical access to the theory of parallel algorithms. In *Proc. ACM SIGCSE'04 Symposium on Computer Science Education*, March 2004.

[11] Russ Miller and Laurence Boxer. *Algorithms sequential & parallel: A unified approach*. Prentice Hall, 2000.

[12] OpenMP Architecture Review Board. OpenMP: a Proposed Industry Standard API for Shared Memory Programming. White Paper, `http://www.openmp.org/`, October 1997.

[13] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[14] Joar Sohl. A scalable run-time system for NestStep on cluster supercomputers. Master thesis LITH-IDA-EX-06/011-SE, IDA, Linköpings universitet, 58183 Linköping, Sweden, March 2006.

[15] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Comm. ACM*, 33(8), August 1990.

How to Construct Small Student Groups?

Tim Heyer
Karlstad University
651 88 Karlstad, Sweden
tim.heyer@kau.se

Abstract

Certain student activities in a computer science education are often performed in small groups from 2 to 4 students. At Karlstad University these activities include e.g. programming activities, writing activities (respondent and opponent), and inspection activities (author and inspector).

Our starting point was a course in Software Engineering where small student groups had first to extend given software models and then to inspect another group's models. Moreover, each student had to write reports and to criticise other students' reports.

Over time, we tried four different approaches to construct student groups for the inspection and the critique: (1) we let the students find partners for the activities, (2) we provided a list of student groups, (3) we used a paper list to let the students create groups dynamically depending on their progress, and (4) we let the students create groups dynamically using a web tool.

The fourth approach is the approach we consider must successful. The tool works basically as follows. Each student registers once and provides name, email address, and telephone number. To join a group for an activity, the student logs in and selects the activity from a list provided by the tool. When a certain amount of students (for example twice the number of students in a single group) expressed their interest to join a group, then the tool creates a number of groups and informs the members by email.

When developing our approach, we tried to achieve the following goals: students groups should vary (being confronted with different approaches to communication, problem solving et cetera should improve learning), students which are ready earlier should be able to proceed earlier (otherwise a student could fail a course because of another student), and the construction of group should require little time from the teacher (the time is better used for teaching than for administration).

The first, second, and third approach above did not satisfied our goals. We found that using a tool as briefly described above did satisfy our goals and we use the tool now in other courses as well.

An Analysis of Gender Impact on Students' Performance in a Written Examination of Software Engineering

By
Kristian Sandahl
krs@ida.liu.se
Dept. of Comp. and Info. Sci.
Linköping University, Sweden

Abstract

The student groups in Software Engineering in Sweden are rather homogenous; most of the students are men of age 19-26. At the department we have for long been aware of that our examination instruments might wrongly discriminate women just because instruments have evolved during a long time and set a tradition maintained from a male perspective only. However, due to the scarcity of female students our knowledge has been limited.

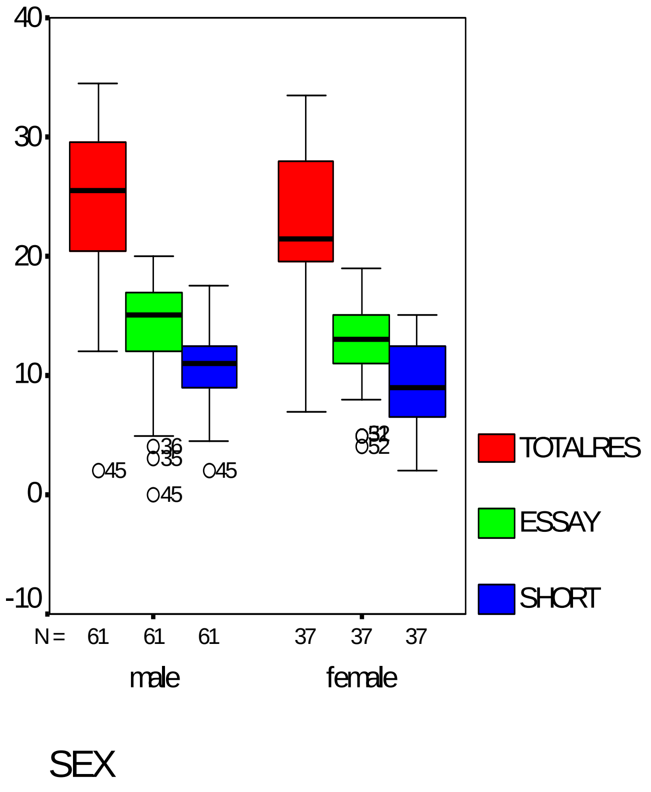
An opportunity for a fair comparison opened in 2004 in the course TDDB62 Software Engineering. The course is given annually for students in the 4th year of the curriculum for Industrial Engineering and Management, especially Computer Science. On top of general courses in mathematics, management and engineering, the students take classes in Computer Science for a year full-time studies. The course is also open to international guest students. This group is very heterogeneous ranging from people with only basic courses in programming to students aiming for a major in Computer Science.

The course is organised in a theory part, a small lab. series and a project, where the students work independently in groups of 6-8 students. The written exam is given in the middle of the course and covers the general theory of Software Engineering only. The exam is divided in a short-question part and an essay-problem part. The short questions are random samples of the entire software life cycle and comprise either questions for facts or small construction tasks. There are 10 questions giving maximally 2 credits each. The essay questions requires longer answers and are made up of larger comparisons between many concepts, larger construction tasks, or analyses of state-of-practice in Software Engineering. There are 5 questions giving maximally 10 credits each. The student selects 2 of the essay questions he/she wants to answer. Maximum credit is 40 the passing threshold is 20.

The distribution of credits for short, essay and entire exam is shown in the box-plot on next page. The thick lines represent the median value, the box represent 25 percentiles below and above the median. The thin lines represent extreme values and circles denote outliers. As can be seen, the median values are higher for male students in all categories. However, the variance is high, so only the difference that is significant is the difference on essay questions. The summary of the Mann-Witney test is appended.

The conclusion is that I as examiner have to be more careful in looking for true facts in essay questions and not be fooled by verbose writing. I also have to be more conscious about this when I present the examination forms during the lectures. Scientifically, this is only one data point that needs further replication to yield general conclusions. I challenge more people to collect similar data.

I'm indebted to Carl Cederberg and Gunilla Mellheden for the data collection.



Ranks

	SEX	N	Mean Rank	Sum of Ranks
SHORT	male	61	52,86	3224,50
	female	37	43,96	1626,50
	Total	98		
ESSAY	male	61	54,26	3310,00
	female	37	41,65	1541,00
	Total	98		
TOTALRES	male	61	53,83	3283,50
	female	37	42,36	1567,50
	Total	98		

Test Statistics(a)

	SHORT	ESSAY	TOTALRES
Mann-Whitney U	923,500	838,000	864,500
Wilcoxon W	1626,500	1541,000	1567,500
Z	-1,504	-2,140	-1,936
Asymp. Sig. (2-tailed)	,133	**,032**	,053

a Grouping Variable: SEX