- two parser generators - a top-down (LL(1)) and a bottom-up (mixed precedence). Only the bottom-up parser generator was operational in the last academic year, however the students practiced top-down parsing by building hand-written recursive descent parsers from extended BNF descriptions of the source languages.

- an almost empty "skeleton" routine for semantic analysis and synthesis. The students try their semantic actions interactively before supplying the code to fill in the "skeleton" routine. The different options that they could explore originate from the different possibilities for defining the concrete syntax and linking it with the semantics of the source language.

- a set of primitive code generation routines that generate assembly code for the TAM virtual machine (see below). The students call these routines from their own semantic programs.

- an error handler, a major part of which is an adjustable syntax error recovery routine implementing "panic" mode of recovery [1]. In addition students are encouraged to implement and explore recovery by replacement and error productions [1].

- a stack-machine emulator. We have adopted the TAM virtual machine described in [9], and have developed an interactive user interface and an assembler for it.

- utility programs. At the time being these include symbol-table, stack and tree manipulating routines.

- a standard interactive user interface that allows prototyping and debugging of compilers being built by the students.

Below we shortly discuss the last three components.

3.1. Defining Symbol Tables

Understanding symbol-table manipulation is a key point in learning contextual analysis and synthesis. Our goal was to implement a reusable and easily adjustable component that would conveniently allow visualization. We chose to implement the symbol table as a heterogeneous C++ class having objects of another class as an attribute. Each symbol table entry consists of a scope level number, a name and an attribute pointer. In this way all symbol-table operations were implemented in advance, while the structure of the attribute was left open. In a minimalist scenario, the students are only expected to specify the structure of their attribute. This was the case in our closed laboratories. In order to make the exercises simpler and to provide a standard visual image of the table, we used attributes with a fixed structure, however the heterogeneous class approach allows much more flexibility. Not only can the attribute be different for different compilation exercises, not only can it be arbitrarily complex (for example, a tree structure), but also each symbol-table entry
can have a different structure for its attribute. This could be useful for more advanced cases or optimal utilization of memory.

An interesting side effect of the flexible implementation of the symbol table was the active interest of several of our students in object-oriented programming.

3.2. Defining and Manipulating Trees

Parse trees and abstract syntax trees are popular and well understood intermediate representations, but it would be unrealistic to expect that the students will have the time to implement tree structures in the time scheduled for our course. Instead we implemented a C++ class that allows flexible tree manipulation compatible with the one described in [Wat-93]. We eliminated the limitation of [Wat-93] on the arity of nodes. For visualization we chose a two-dimensional representation of trees as more appropriate for our educational goals, instead of the widely used linear representation.

3.3. The Stack Machine

The code generation part of a compiler is very much dependent on the target machine. Different machines have different addressing conventions and register configurations. A virtual stack target machine has many advantages for a first course in compilers, like ours. Firstly, using a stack target machine eliminates the problem of intermediate storage and register allocation for expression evaluation. Secondly, the stack is the natural run-time storage organization for languages with nested scopes and recursive routines. Both texts [Wat-93] and [Aho-86] refer to virtual stack machines, but the presentation in [Wat-93] is tightly bound to such a machine. We found this presentation very useful and easily understandable by the students.

The TAM stack machine [9] has two separate stores for code and data. The data store accommodates a stack and a heap growing in opposite directions. All evaluation takes place on the stack. Primitive arithmetic, logical and other operations are treated uniformly with pre-programmed functions and procedures. An important advantage for our course is the fact that the procedure call and return conventions are implemented at the TAM machine level. This simplifies code generation, while still allowing students to observe the adopted run-time storage organization. A number of registers are dedicated to specific purposes.

Our implementation of the TAM stack machine includes a loader and an interpreter. Each of them illustrates a topic in our course. The interpreter can be run in a step-by step mode, thus allowing students to observe the execution of their compiled programs. Fig. 2 is a snapshot of a screen showing our user interface to the TAM interpreter. The current instruction is at address 132 of the code store (pointed by the register CP). The program is stored from location 0 (register CB) to location 199 (register CT). Next available location in the stack is 21 (register ST), the stack base is 0

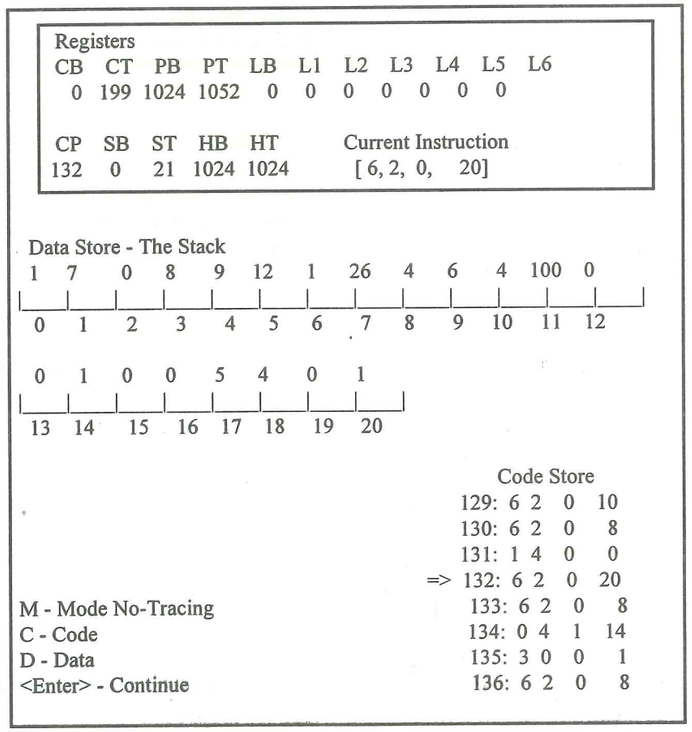

Fig. 2. A snapshot showing the TAM interpreter interface

(register SB). The heap is empty (HB = HT = 1024). We are executing the global part of the program (LB=0, and all link registers - L1 to L6 contain zero). The primitive routines addresses are stored from location 1024 (PB) to location 1052 (PT).

We also implemented an assembler for the TAM machine. There were several reasons for this implementation.

- Our syllabus includes coverage of assemblers as simple translation systems. The forward reference problem and its solution via backpatching are illustrated on the example of assemblers. We also use assemblers to illustrate languages with monolithic structure.

- We cover assemblers before code generation, and use the TAM assembler to introduce the TAM machine itself.

- Our code generation routines generate mnemonic TAM assembler code rather than numerical TAM machine code. This greatly increases the readability of the generated code and helps the students understand the code generation templates for control structures and expression evaluation, as well as identify the possibilities for optimization. As forward references are dealt with by the assembler, this also simplifies the contextual analysis and allows the students to concentrate separately on its other aspects - identification for nested scopes and type checking.

As a bonus, the TAM assembler and interpreter will be used for illustrations in two other courses - Machine Organization and Assembler Programming.

3.4. The Compiler User Interface

As mentioned above, our decision was to make a learning, rather than teaching toolkit. The process of learning compiler construction involves prototyping the behaviour of the compiler, as well as locating and repairing errors in compilation. The location of a defect may not be obvious from the generated target code. The user interface must allow the student to study the behaviour of the compiler in order to find the source of a defect.

Our toolkit views the student as the active and only participant in the learning process. It provides a platform from which the student can actively explore and control the interactions and monitor the data and control flow in the compiler being built. The student can interact with the various parts of the compiler, examine and change data values at any time. The feedback provided by the interface helps the student understand where he is in the compilation process. There are several paths of interaction between the toolkit and the student:

- step by step output of the canonical parse on the screen and in a listing file. This includes the production applied and the current sentential form at each step of the parse.

- screen output of the contents of all stacks. The students can change the stack values at each parse step. This is particularly useful when prototyping semantic analysis and synthesis routines which manipulate stacks parallel to the syntax stack.

- screen output of the symbol table. After examining its contents, the students can change it as appropriate.

- direct emission of code or comments in the object file. Students can enter the code to be emitted from the keyboard.

- screen output of the generated code and data. At each step of compilation the students can view and if necessary change the code and/or the data generated so far.

4. The Laboratory Component of the Course

Designing the laboratory component for a course is not that different from designing the course itself. You need to have in mind who you are teaching, what you are teaching, and how you will teach it. The stress in our course is on simpler concepts that can be mastered in the limited scheduled time. We take into account the difficulties our students have with advanced theoretical material. We try to support the lectures with numerous examples, case studies and practical work.

There are 14 closed practical exercises included in the course. Here we will mention only the most important of them. An essential assumption for the success of each laboratory session is that students know in advance their tasks and do some minimal theoretical preparation for them.

4.1. Scanning with Case-Type Scanners

This is the first scanning exercise. Its purpose is to familiarize the students with the standard scanner interface and with the structure of the case-type generated scanners. They define a language in BNF, construct a scanner for it, and interface the scanner to a main program (e.g. to output each source line in reverse order, search for given words, print the source program with indentation depending on certain tokens etc.). The students design their own main program or choose its functionality among several suggested ideas.

4.2. Canonical Parse and Synthesis

This exercise is done in the fifth week of the semester. It resembles an example done in class. The students study a very simplified compiler that still generates meaningful code. The purpose is to illustrate the syntax driven nature of the compiler and to show how using an additional stack the compiler can keep track of the run-time location of operands and intermediate results in expressions. The corresponding ideas are central in the course. They are introduced in the preceding lectures and recur and are further developed throughout the semester.

4.3. Assemblers. Simple Languages with Monolithic Block Structure. Backpatching

The TAM assembler is a second example of a simple translation system built with our toolkit. The students study its structure, follow the backpatching process, and get acquainted with the TAM abstract machine and its user interface. Sample assembler programs for searching and sorting arrays are provided, but students are expected to write their own examples.

4.4. Syntax Error Recovery with Top-Down and Bottom-Up Parsing

This is an important exercise in which the students try to limit the impact of source program syntax errors on the syntax analysis. They design sets of stop-symbols for so called "panic" error recovery and experiment with them. They verify the statement that the set of stop-symbols is very much language dependent and blindly including too many stop-symbols has an equally negative effect as including too few of them. Students are encouraged to use the second half of the scheduled time to include in the syntax of their language error productions or to experiment substituting an expected symbol for an erroneous one.

4.5. Semantic (Contextual) Analysis

This exercise illustrates the concepts of identification and type checking introduced in the lectures. A skeleton structure for the compiler is provided, as well as sample identification and type checking routines, and hints how to use them. After constructing their semantic analyzer, the students are expected to test it with correct source programs and with programs containing some typical errors, such as undeclared identifiers, double declarations in the same scope, type mismatch in an assignment, type mismatch in an arithmetic or boolean expression, missing return statement in a function body, parameter type mismatch in a function call, etc.

4.6. Run-Time Storage Organization for Languages with Nested Block Structure

Two exercises are devoted to this topic. The source language has only declarations and assignment statements with no arithmetic or other operations. The purpose is to concentrate on run-time representation of different data types, stack storage allocation, and parameter passing protocols. The students experiment with the compilation process and observe the behaviour of their compiled programs on the TAM stack machine.

4.7. Code Generation

There are two exercises for this topic. The first one uses a simple language with monolithic structure, expressions with arithmetic and logical operations, assignments, branch and loop statements. Auxiliary routines for building code templates are provided. The students are expected to incorporate these in the compiler. The second code generation exercise is the last for the course. It integrates the code generation with all the other techniques learned during the semester. The students use a simple, but full featured source language.

5. Conclusion

There are several factors of Computer Science education that make the application of our toolkit feasible. The most important one is perhaps the necessity of supporting the theoretical considerations in the concrete subject area with numerous practical examples. These make the specific knowledge more digestible for students and give them the feeling of quantitative evaluation of theoretical considerations. The other factor is the need for the students to apply the Software Engineering knowledge acquired in a course that is taught just a semester before the Compilers course. Building a complete compiler for a non-trivial language can constitute a term project for a team of students and certainly provides an exiting software engineering experience. It is worth mentioning that our toolkit itself is being implemented mainly through individual and group student projects.

We have recently obtained several tutor suggestions for extensions of our toolkit. The main ones are: a log of the student interactions with the system and a help dialogue when no interactions are made. We intend to use a weighted approach to asses the useability of the toolkit's components and decide on the further directions for improvement.

6. References

[ACM-91] Computing Curricula 1991. Report of the ACM/IEEE-CS Joint Curriculum Task Force. ACM Press, IEEE Computer Society Press 1991

[Aho-96] Aho, A.V., R. Sethi, J.D.Ullman, Compilers. Addison Wesley 1986

[Ben-90] Bennet, J.P., Introduction to Compiling Techniques, McGraw Hill 1990

[Cou-93] de Coulon, F., What Can We Really Expect from Computer Aided Learning in Engineering Education. In: Proceedings of International Conference on Computer Aided Education CAEE 93, Bucharest 1993. Ed. D. Ioan, p.3-8

[Hol-90] Holub, A. I., Compiler Design in C. Prentice Hall 1990

[Mar-95] Maredi M., H.J.Oosthuizen, A Problem Solving CAI - Factor Q. Computers and Education, Vol. 25, No 4, December 1995

[Sch-93] Scherbakov N., in oral contribution at the International Conference on Computer Based Learning in Science, CBLIS 93, Vienna, 18-21 December 1993

TEACHING TURING MACHINES - LUXURY OR NECESSITY?

Y.Velinov
Department of Computer Science and Information Systems
University of Natal, Pietermaritzburg
yuri@cs.unp.ac.za

Abstract

The purpose of this paper is to outline an approach for the development of the Tape Machines model in the Theory of Computation which is bound more closely to the real computer world. The approach is based on a developed software consisting of a Tape Machine Simulator and a Tape Machine Assembler. It gives possibility to achieve the aims of the theory without any loss of mathematical rigor but in more natural, useful and attractive manner.

Introduction

Tape Machines were introduced by A. Turing [1936] as a theoretical model of the concept of computation several years before the real electronic computers appeared in practice. The hypothetical devices originally considered by A. Turing (Tape Machines known now as Turing Machines) are appealing because of their simplicity, transparency and completeness. On one hand, they embrace all aspects of the algorithmic processes (the concept of data, the concept of a control device, the concept of an algorithmic language), and on the other hand they convey a strong feeling of a mechanistic device working by itself and independent of any human intellectual activity.

Nowadays, Tape Machines in their different forms are an indispensable pedagogical vehicle in contemporary courses in the Theory of Computation. However, they and the ways the theory around them is developed suffer some disadvantages. The first thing to note is that the original Turing machines are extremely clumsy when really interesting computations have to be performed. There are explicable reasons for that. From a purely theoretical point of view any model of computation must face two discrepant requirements:

- The model must be as simple as possible in order to be convincing as a mechanical approach and also in order to be easy to simulate it by other models of computation.
- The model must have as much expressive power (not just computational power) in order to deal easily with the real computations and also in order to be able to simulate in an easy way other models of computation.

Every model of computation balances these two requirements. Initially the first requirement was considered more important (mainly for the model to be convincingly mechanistic) and the original Turing machines were accepted as more sound. Typically, the theory developed around them consists of descriptive languages for building Turing machines and series of lengthy (and I dare say, cumbersome) theorems proving step by step what can be computed or modeled, where in the heap of details it is quite easy to lose the essential ideas. Later on, sacrificing simplicity to gain more convenience some other machine models, which move the balance from the first requirement to the second one, were developed [Shepherdson, Sturgis 1963, Minsky, 1967]. Regardless of their attractiveness I still find the models unadequate for a very simple reason: developed historically in the frame of pure mathematics and following its traditions, these models and the theory around them are a successful, but not very comfortable attempt to build the theory of computation outside of the existing nowadays in reality computing phenomena. As a result the courses in the Theory of Computation in the Computer Science syllabi are frequently perceived as stand alone impractical outsiders which are included there only to show some respect to the achievements of the mathematicians before the real computer science came to life. This unsatisfactory state was discussed before (see for example [Brady 1974]) but looking at the nowadays textbooks I still cannot see any significant changes in this respect.

The purpose of this paper is to outline an approach for the development of the theory of Tape Machines which is bound more closely to the real computer world. I will try to show that without any loss of mathematical rigor the aims of the theory can be achieved in a more acceptable and natural manner. That if we wish we can even rise the level of rigor. Finally, that the theory can outline ideas used in the real computer practice.

Tape Machines

If the first of the requirements described above is regarded as essential the initial machine model must be as simple and as transparent as the original Turing Machines are. This together with the fact that writing programs is considered as the most fundamental task in Computer Science makes the machine proposed by Wang [1957] a natural candidate. In the further considerations I will accept as a groundwork the modification of the Wang's machine which can work on arbitrary symbols.

A *Tape Machine* (fig.1) consists of three components: a data tape, a program tape and a processor.

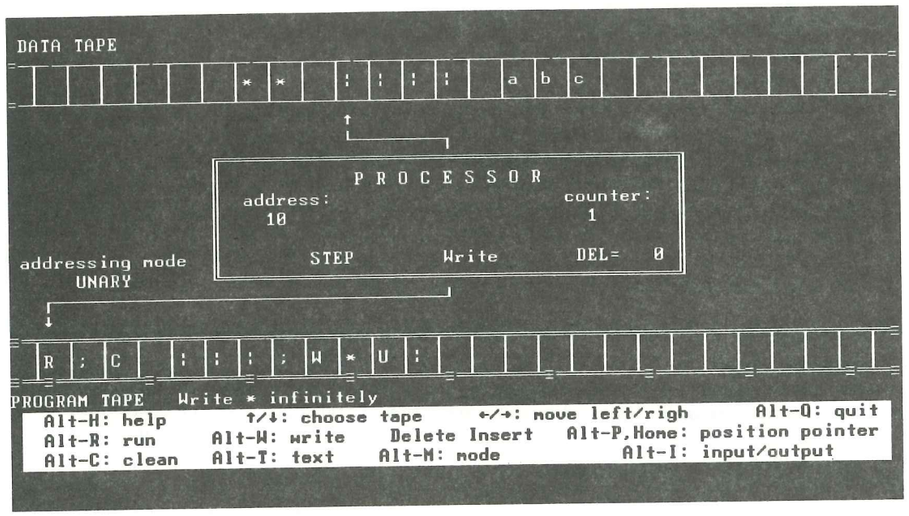

Fig.1.

The *data tape* is infinite in both directions. Each of its cells can be empty (_denotes the empty symbol) or can contain exactly one symbol from a fixed alphabet $\mathcal{A}$. Data, properly organized, are situated on the otherwise empty data tape symbol by symbol. When the machine works the size of the area which contains nonempty cells may change but is always finite.

The *program tape* has a beginning and is infinite in only one direction. Each of the cells of the program tape can contain exactly one of the symbols from the alphabet

$$\{\text{'R'}, \text{'L'}, \text{'W'}, \text{'C'}, \text{'U'}, \text{'|'}\} \cup \mathcal{A} \cup \{\_\}.$$

The purpose of the program tape is to store the instructions for the processor - the *machine commands*. The machine commands are the symbols 'R', 'L' and the strings from the sets $\{\text{'U'}\}\blacklozenge\{\text{'|'}\}^*$, $\{\text{'W'}\}\blacklozenge\mathcal{A}$ or $\{\text{'C'}\}\blacklozenge\mathcal{A}\blacklozenge\{\text{'|'}\}^*$ (here $\blacklozenge$ stands for the concatenation operation). We call the symbols 'R', 'L', 'W', 'C' and 'U' the command codes; U is the code for unconditional jump, C is the code for conditional jump, W is the code for write operation, R is the code for right movement and L is the code for left movement. Only a finite number of cells in the programming tape can contain nonempty symbols.

A continuous sequence of machine commands is called a *machine program*. A machine program must be situated on the program tape symbol by symbol starting from the first cell, with no empty cells between the commands. All unused by the program cells on the program tape must be empty.

The commands in a program are naturally numbered in the order in which they appear with consecutive positive integers starting from 1. The ordinal numbers of the commands in a program are used by the processor when implementing the jump commands.

The *processor* works in discrete time. The time scale is represented by the sequence of the natural numbers. At every moment of its work the processor observes exactly one cell of the data tape and one cell of the program tape. It can move the observation point on the data tape one cell to the left or to the right or it can write in the observed cell any symbol from $\mathcal{A} \cup \{\_\}$. It can also move the observation point on the program tape one cell to the right, or one cell to the left if the observed cell is not the first one. The processor is organized in such a way that it follows the commands written on the program tape. At the beginning the processor observes the first cell of the program tape. If it has just started working or it has just executed a command the observed cell on the program tape contains a command code. The processor analyzes the command moving if necessary the observation point on the program tape to the right cell by cell until next command code is reached. If a syntactically well formed command is recognized the processor interprets it implementing some actions according to the type of the command as follows:

- If the command code R is observed at some moment then at that moment the processor moves the observation point on the data tape one cell to the right and moves the observation point on the program tape one cell to the right to analyze the next command at the next moment.
- If the command cod L is observed at some moment then at that moment the processor moves the observation point on the data tape one cell to the left and moves the observation point on the program tape one cell to the right to analyze the next command at the next moment.
- If the command code W is observed at some moment the processor moves the observation point on the program tape one cell to the right to see the symbol à situated there, then it changes the content of the observed cell on the data tape to à and once more moves the observation point on the program tape one cell to the right to analyze the next command on the tape at the next moment.
- If a command code U is observed the processor analyzes the content of the following cells one after the other until a new command code is encountered. In the event of a syntax error the processor stops. If a syntactically correct command - U with *n* symbols | after it, has been recognized the general reaction of the processor to the command (we do not specify in details how it reacts at each moment) is to move the observation point on the program tape at the beginning of the *n*-th command to process it at the next moment.
- If a command code C is observed the processor analyzes the content of the following cells sequentially until a new command code is encountered. In any case of syntactical incorrectness the processor stops. If a syntactically correct command - Cα with *n* symbols | after that, has been recognized the general reaction of the processor is as follows:
  - in the case when à is observed on the data tape move the observation point on the program tape at the beginning of the *n*-th command;
  - otherwise move the observation point on the program tape to the beginning of the next command.
- The processor stops working if at the very beginning or just after the implementation of any command a symbol other than a command code is observed.

One more possible step in order to make the Tape Machine more sound and correlated with the other topics in Computer Science could be to describe the processor as a finite automaton. Following this direction it deserves also to consider the relations between the Tape Machine and the original Turing Machines. The "program" part of a Turing Machine is nothing else but a description of a finite automaton, and we usually consider finite automata as hardware structures. It is also possible to show that each program for a Tape Machine can be translated into a program for a Turing Machine and vice versa. This underlines the important fact that the programs (with no recursive calls) are equivalent to the hardware constructions.

The imaginary Tape Machine described above can be brought to life in the form of a simulator on a real computer. Fig.1 shows the appearance of the screen of a simulator implemented for the IBM-PC computers.

Assembler for Tape Machines

Having a simulator at hand it is easy to demonstrate how inconvenient it is to write and change machine programs. This naturally justifies a next step - the introduction of an assembly language and an assembler for the Tape Machine. A simple form of an assembly language for the Tape Machine is described below.

A program written in the assembly language for Tape Machines consists of a sequence of commands each one written on a separate line. The commands have a standard format consisting of 5 fields of fixed length though some of them can be empty:

LABEL	CODE OF OPERATION	SYMBOL	LABEL/NUMBER	COMMENTS
4 symbols	3 symbols	1 symbol	4 symbols	15 symbols

In order to identify command lines (necessary for the conditional or unconditional jump operations) the assembly language uses strings of symbols called "labels". A label is a string which begins with a character and contains no more than 4 characters. The LABEL FIELD may contain any label but no two different lines may have the same label in it. A label in the label field of a line identifies the command in it. The SYMBOL FIELD may contain any symbol accessible on the keyboard of an ordinary computer. The CODE OF OPERATION FIELD may contain any of the strings 'LFT', 'RGT', 'WRT', 'BEQ', 'BNE', 'JMP', 'HLT' which are the operation codes. The ADDRESS/NUMBER FIELD may contain an label or a four digits positive integer number. The COMMENTS FIELD may contain any sequence of symbols. It is not significant for the operation of the program an can always be left empty.

The content of the different fields of a command line depends on the used operation code. The meaning of the operations together with the format they require is described bellow, where '*nnnn*' denotes a four-digits number, '*mmmm*' and '*llll*' - labels, '*x*' - a symbol, and '_' - an empty space for a symbol:

mmmm LFT _ *nnnn* - Move the observation point on the data tapes *nnnn* cells to the left.

mmmm RGT _ *nnnn* - Move the observation point on the data tapes *nnnn* cells to the right.

mmmm WRT *x* ____ - Write the symbol x on the data tapes.

mmmm BEQ *x* *llll* - If the observed symbol on the data tape is x continue the execution of the program with the command labeled by *llll* or otherwise continue with the next command.

mmmm BNE *x* *llll* - If the observed symbol on the data tape is not x continue the execution of the program with the command labeled by *llll* or otherwise continue with the next command.

mmmm JMP _ *llll* - Continue the execution of the program with the command labeled by *llll*.

mmmm HLT _ ____ - Stop the execution of the program.

The commands of the assembly language are executed one after the other in the natural order (from top to bottom) of the lines they occupy, except when commands with code of operation JMP, BEQ or BNE are encountered. Such commands change the natural order of execution of the commands according to the meaning of the operations as it was described above.

It is necessary to show further that the programs written in the assembly language can be translated into the machine language. Instead of giving a formal proof of this fact it is better to construct an assembler. A two pass assembler seems more simple and convenient for the purpose. The assembler itself should not be constructed on a tape machine. It should be considered only on algorithmic level and of course could be designed in a high level language. The algorithm for its work stands for a proof of the translatability. Moreover, the proof of the translatability can be supported further with a proof of the correctness of the assembler algorithm. Fig.2 shows the appearance of the screen of an implementation of the described assembler for the IBM-PC computers.

Though applied to a very simple assembly language, except for the subroutine calls which cannot be introduced at this stage, the assembler for Tape Machines covers the most significant features of a real assembler. For a more deep understanding of the assemblers structure, if this is regarded as essential, a more rich assembly language with more commands can be introduced. Macrodefinitions and even more - relocatable code together with linkers and loaders may also be considered.

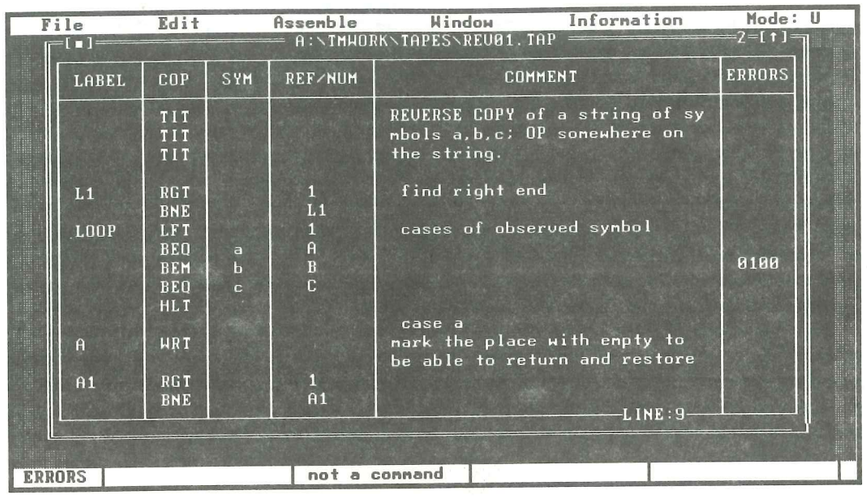

Fig.2.

Assembly language level is not considered in the standard texts concerning Turing Machines. Usually, only remarks that without really increasing the computational power of the machines some additional commands can be introduces are present. The introduction of an assembly language and an assembler not only relieves the exposition of the theoretical results but also combines the needs of the theory with the practical aspects of Computer Science.

Register Tape Machines

The topic can be developed further by introducing a higher level language such as the language of flow diagrams considered by Hermes [1965], but it is better to continue in another direction - using the assembly language to design a virtual Register Machine in order to incorporate the approaches of Shepherdson-Sturgis [1963] or Minsky [1967].

A Register Tape Machine uses unlimited but finite number of registers instead of cells. Each register itself is a tape infinite in one direction, which can contain an infinite chain of symbols. The registers can be simulated on the data tape of the ordinary Tape Machine with the aid of two additional servicing symbols, say '#' and '$'. The symbol # is used to separate the different registers and $ - to indicate the beginning and the end of the field of registers on the data tape. On the data tape the registers are situated attached, one immediately after the other. They can be distinguished by their ordinal numbers or by appropriate names associated with them. The next figure shows a data tape with 2 registers containing some data.

Since each of the the registers must be potentially infinite if a symbol is to be inserted in a register at a certain place a free cell is created there by shifting all the cells till the end of the register field one

cell to the right. If a symbol in a register is to be deleted at a certain place all the cells from that place to the end of the register field are shifted one cell to the left. Therefore, though implemented by a single infinite tape, the virtual Register Machine appears to the user as a list of infinite tapes - registers each one distinguished by a name or a number.

The language for the Register Machine can be constructed directly to be of assembly type. The commands could be of the same format as the format for the commands of the assembly language for Tape Machines. The minimal set of commands must include a command of the type DR *xxxx* (define register) used to introduce a new register on the data tape and associate it with the name '*xxxx*', commands for attaching a symbol at the left (right) side of a register and conditional jump commands depending on the first observed symbol in a register (or distinguishing an empty register). More complex commands can be introduced to compare the content of any two registers. Furthermore, the natural numbers can be represented in unary code as sequences build using a fixed symbol and stored in a register. Then commands for arithmetical operations can be considered or simulated. At this stage all significant commands can be implemented by appropriate sequences of ordinary Tape Machine commands to justify the claim for virtuality of the Register Machine. Further, in order to make the virtual Register Machine more convenient, in analogy with the real computers, some standard resisters can be introduced. For example an accumulator and a stack. The stack gives further possibility to introduce subroutines, a mechanism for recursive calls, mechanisms for passing parameters and so on. A higher level language can also be introduced if necessary.

With the virtual Register Machine at hand the development of the topic can continue to reach the typical goals of the theory of computation (like proving the equivalence of different models of computation, or establishing validity of important theorems like the Theorem of the three indexes or the Theorem of Recursion) but in different stile - by designing programs instead of by proving theorems. And the level of rigor can be raised by proving the programs correct.

Conclusions

Let us summarize what can be achieved if the approach described above is followed.

First of all, on the expense of building several levels of languages and machines, and considering their interrelations the theoretical results can be achieved more easy. The formal troubles in most of the proves are shifted in advance on the constructions of the assembly language or higher level languages and their translation to the machine language. As a result the proves themselves become more clear and communicate better the underlying ideas. This can be expected - the introduction of several levels of languages reflects the popular in the Computer Science methodological principle "divide and conquer" developed there under the pressure of the problem of solving real complicated problems.

Further, the stile of the proofs is unified and consists of construction of programs and proving them correct.

Finally, building an assembly language and the corresponding assembler, constructing a virtual machine and a language to deal with it is beneficial by itself for the computer science education. The ideas of how an assembly language is designed and how an assembler is constructed are developed in a simple way and become more clear and understandable. This gives opportunity to introduce them at the early stage of education.

As a whole this approach brings nearer the theory and the practice in the Computer Science to the benefit of both. But there is also another reason supporting it. The variety of ready available high level languages on one hand and the complexity of the contemporary real assembley languages on the other lead to a decreasing interest in studying the assembley language level. There is a tendency to elude assembley language courses in the software directed syllabi. As a result the students lose the touch with the low level programming. The proposed approach is an alternative which gives possibliliy to incorporate it into the theoretical courses.

The ideas described above were experimented quite successfully, at different stages of their development, with first and second year students. As it could be expected the introduction of simulators made the topic much attractive. The students enjoyed playing with the software supporting the course and prefered it to the blackboard presentations. But playing they succeeded to rich faster more deep understanding of the material.

References

Brady, J.M.,
A Programming Approach to Some Concepts and Results in the Theory of Computation, The Computer Journal, V19, N3, p.234, (1974)

Hermes, H.,
Enumerability, Decidability, Computability., Springer (1965.)

Minsky, M.,
Computation: Finite and Infinite Machines., Prentice-Hall, (1967).

Shepherdson,J.C., Sturgis,H.E.,
Computability of Recursive Functions., JACM,Vol.10, p.217-255 (1963).

Turing, A.,
On Computable Numbers With an Application to the Entscheidungsproblem., Proc. London Math Soc. Ser.2, 42 (1936).

Wang Hao,
A variant to Turing's Theory of Computing Machines., JACM, 4, 1 (1957).

LESSONS LEARNT FROM USING C++ THE OBJECT-ORIENTED APPROACH TO SOFTWARE DEVELOPMENT

Dr Rose Mazhindu-Shumba
Computer Science Department
University of Zimbabwe
Box MP167 Harare
Zimbabwe: email:mshumba@zimbix.uz.zw

Abstract

This paper presents the lessons learned from using the C++ programming language and an object-oriented design methodology for the development of a prototype analysis tool for inheritance relationships in object-oriented C++ software systems.During the project two tools were developed; a data extraction tool, and an analysis tool. We noted that inheritance and dynamic binding , reuse and strong typedness are powerful features of C++. However we also noted that; C++ is an evolving language which needs standardization, the learning curve is very long, C++ leaves the programmer to resolve most problems and C++ code is very difficult to navigate and understand.We hope that the lessons explained in this paper will not only enhance the understanding of the C++ language, but also be very useful to anyone who intends using the language with any other object-oriented designing approach.

Introduction

The C++ language Stroustrup [Str86] has become one of the most popular languages for the development of object- oriented software. We feel that the popularity of C++ is due to the following facts:

C++ is derived from C, itself a popular language. Therefore C developers looking for the benefits of object- oriented programming look to C++ as a logical step towards object-oriented programming. C programmers move incrementally to C++, adding in object-oriented techniques at their own speed.

There is plenty of literature in the form of books and journals that explain and illustrate the use of the C++ language.

There is many development tools for the C++ language.

The C++ language is available on most platforms;Unix, Dos and VMS.

In this paper, we explain the lessons that we have learnt from using C++ and an object-oriented design methodology (the Rebecca Wirfs-Brock et al, [WBWW90]) in the development of an interactive computer-aided prototype analysis tool for inheritance relationships in object- oriented C++ software systems, SAOOSS (A System for the Analysis of Object-Oriented Software Systems). SAOOSS consists of a set of tools; a data extraction and storage tool (DEST) to extract required data from a given C++ system and store it into an object-oriented database and an analysis tool to analyse the data in the database. SAOOSS was implemented on the Unix platform using the Sun 2.1 C++ compiler, the OSE C++ third party libraries by Dumpleton, [Dum93], and a graphical interface development toolkit, tcl-tk by Ousterhout [Ous93]. In all a one hundred classes were designed and implemented.

We hope that the lessons explained in this paper will not only enhance the understanding of the C++ language, but also be very useful to anyone who intends using the language with any other object-oriented designing methodology.

Lessons learnt from using C++

Inheritance and dynamic binding a powerful feature

From the development of SAOOSS, we found that polymorphism, inheritance and dynamic binding together form a powerful feature of C++. As part of (DEST), we have some classes as presented below

an abstract class DETAILS with a pure virtual function get-details(),
classes INHERITANCE-DATA and METHOD-DATA publicly inheriting from class DETAILS and client C++ code given below the classes.

In C++, a pointer to a base can point at either the base class object or a derived class object. The member function selected depends on the classes of the object being pointed at. The implication of the statement: p[i] -> get - details() is the following:

Invoke at run time the get-details() member function, which corresponds to the derived type of the object that p[i] is pointing to. If the object's class has no override for get-details(), use the parent class get-details().

The power of inheritance and dynamic binding

```
class DETAILS
{
public:
virtual double get-details()=0;
};

class INHERITANCE-DATA: public DETAILS
{
double height, width;
public
double get-details();
};

class METHOD-DATA:public DETAILS
{
double radius;
public
double get-details();
};

client C++ code
DETAILS *p[N];
for (i=0;i<=N;++)
    get-details=p[i]->get-details()
```

When the client code is executed, the selection of which method to invoke for a virtual function is dynamic. The advantage of such a design that uses inheritance and dynamic binding is that is that, if more classes that represents data to be extracted from given system are added to the inheritance hierarchy, our code needs not change. This helps create open-ended systems.

Reuse support in C++ is a powerful feature

In C++, code reuse is achieved by use of aggregation, templates and private inheritance. We found aggregation and templates very useful in the development of SAOOSS.

Aggregation

Aggregation is the containment of an object of a class inside another class. There were several cases in the development of the analysis tool where we only needed to reuse a small percentage of the parent methods. in such a case it was less appropriate to use public inheritance since the is-a-kind of relationship did not exist among the classes. Since what we wanted was reuse of some functionality (not all), aggregation was used. Although we could have used private inheritance, we felt aggregation was better. Aggregation preserves the encapsulation of the contained objects, whereas when private inheritance is used, the inheriting client can have access to protected members of the base class. The fact that aggregation preserves encapsulation means that it reduces coupling among classes. Two objects are coupled if and only if at lease of them acts upon the other. Any evidence of a method of one object using methods or instance variables of another object constitutes coupling Chidamer and Keremer [CK94].

Templates or parametric polymorphism

We found templates to be very useful in the development of SAOOSS. all the classes in the OSE C++ data structure library are templated. Whenever we wanted a specific type of data in the structures, all we needed doing was instantiate the type. This saved us a lot of development time and effort and helped produce consistency across the developed system. However, the current problem with templates is that most available compilers do not support them as yet, since they are a new feature recently introduced into C++.

Strong typedness is a powerful feature of C++

C++ is a strongly typed language. A strongly typed language is one in which it can be guaranteed that all expressions have correct types at compile time. The strongly typedness of C++ is due to the fact that it has many fundamental types, it lets programmers define new types and has more restrictions on the operations allowed on the instances of the type. C++ checks the message names and types of arguments in a class function call at compilation. The massage must be defined for the object class. In addition all the arguments must be types derived from the types declared for the function's arguments.
The importance of checking types at compile time is that in object-oriented programming, member functions can be misused. The misuse of a message may not be obvious when a program's source is checked. In addition since inheritance distributes the functions defined for a class, it may not be obvious whether a function is available to a particular class.

The C++ learning curve

Learning the C++ language is not very straight forward, the syntax of the language is not very consistent, especially when coming from a Pascal background. The learning of the syntax often distracts one from more important issues like the use of the object-oriented features.

Once the syntax is mastered the next step is to learn how to use the object-oriented features of C++. Learning to use the object features of C++ requires climbing a long learning curve. It takes several months to really appreciate the power of C++ classes and to become class designers. In fact it takes much more time to come to terms with the technical features of object-oriented programming than to move to another procedural language. We feel that it is very difficult to use the object- oriented features of C++ because C++ is a general purpose language that supports object-oriented programming among other things. In addition C++ class constructs come with a variety of protection mechanism, virtual function, virtual base classes, const member functions, reference variables. These features place a heavy burden on programmers trying to move to C++.

We feel that learning C++ and object-oriented programming is an ongoing process which will never end since C++ is continuously evolving. 2.5.C++ is an evolving language, which needs standardization

From our experiences of using C++ and reading other people's experiences, we feel that there is a very urgent need for the standardization of the C++ language. No doubt C++ is perceived as one of the languages that is best suited for modern software development. At the moment there is great variety of C++ implementations. Many of the available implementation produce their own interpretation of some C++ features. We feel that in the absence of real standards for language definition, future implementations will inevitably lead to multiple dialects. There is great need for established conventions and standards for design documents and coding.

Many development projects that find C++ to be the most appropriate language for their needs do not choose to utilise it because of the instability of the language definition, Lenkov [Len92]. C++ is a language that is still evolving and changing too rapidly to be sage choice for large software development.

Since we started working with C++ in 1990, several new features were added to the language by Standard Committee. These include templates, exception handling and rules for relaxing the return types for virtual functions.

C++ leaves the programmer to resolve most problems

The C++ language assumes programmers know what they are doing and the available language constructs should make it easy to do what is desired.

Programmers have to figure out improper overriding of member functions

Whenever a virtual function is declared in C++, the overriding function must have the same signature as the virtual function in the ancestor. If the overridden signature does not match the signature of the virtual member function in the ancestor class, C++ treats the member functions as two distinct member functions with no relationship to each other. C++ leaves the programmer to figure out where a member function is not properly overridden. We feel that C++ should at least warn us or report this as an error. The consequences of leaving such a resolution to the programmer, can result in unpredictable results.

Mistakes in overriding a member function

```
class NUMBER                        class BIGNUMBER: publicNUMBER
   {                                {
   public                           public
   virtual void add (int x)         void add(int x, int y=10)
   {                                {
     return x*x;};                  return x*y;};
   };                               };
    void main()
   {
    NUMBER *p=new BIGNUMBER
    cout <<p -> add(3) <<endl;
   };
```

Support we have a parent class NUMBER, child class BIGNUMBER and a main function as shown above. The reader of the program may expect 30 as the printed answer. We can justify this in the

following way, 'add' is virtual function NUMBER. In function main, a pointer to an instance of class BIGNUMBER is substituted for the pointer to class NUMBER. The function 'add(3)' is then invoked through the object pointer p. We would expect the add function in the child class to be invoked to produce an output 30. However the redefined function 'add' in BIGNUMBER violates the requirements for overriding a member function. The signature of the function does not match the signature of the member function in the parent class. Therefore static binding is used and the version of 'add' in parent is bound to the object of type child, producing an output of 9.

Programmers have to choose which member functions to make virtual and which to make ordinary

C++ leaves the programmer to choose which functions are to be dynamically bound and which functions are to be statically bound. How can a designer of a class hierarchy, for example in a class library know in advance which functions some clients intend to requiredynamic binding? Some researchers have proposed that it might be wise to declare all C++ functions as virtual Coplien [Cop92], Weiner [Wei94]

Programmers have to choose which ancestor classes tomake virtual

C++ resolves the repeated inheritance problem by using virtual base classes. A designer of a class library is again forced to make decisions about which classes to make virtual. Once a class is make virtual, it cannot be undone by a client who wishes to create a class which has two copies of the ancestor. The classhierarchy designer actually imposes decision about the sharing in the descendant class. .

Programmers have to resolve member function and datamember name conflicts.

Whenever multiple inheritance is used, it is possible that there maybe member functions and data member name clashes from multiple ancestors. C++ leaves the programmer to resolve such clashes. The user has got to add code into the child class to resolve the clashes.

C++ emphasizes user defined memory management

A memory-safe language is one which a user's program cannot unknowingly corrupt the contents of memory Reed and Wyant [RW92]. C++ is defined in such a way as toprevent the compiler from statically checking memory operations. One of the most undetected programming errors is the index outside an array bound. If the index is large enough the operating system may signal a fault else the erroneous index goes undetected. The value may later be passed into other parts of the program and some data structures maybe corrupted. In addition, C++ objects can be created using the new operator. The programmer has to figure out when it is safe to deallocate an object. The only time it is safe to deallocate an object is when there are no more references to it. As soon as an object is deallocated the C++ language runtime routine makes the object available for a subsequent instance allocations. We end up with a situations where two different sections of a program are using the same piece of storage to represent two entirely different objects. This causes program crashes. The way C++ handles memory management is in contrast with other object-oriented languages such as Smalltalk and CLOS. In Smalltalk and CLOS, unused objects are automatically reclaimed by the system. The programmer is then free to worry about more substantial issues.

The fact that C++ leaves most problems to the programmer may seem a good idea. A programmer is left with the maximum flexibility of expression. From our experience, this is bad in that programmers are too often tempted to select poor tradeoffs because they do not understand the multitude of options available.

It is very difficult to navigate and understand C++ code

C++ is an operator intensive language. Many operators maybe used with different meanings in different context. For example the operator `=' can be used to mean initialise, assign or establish a pure virtual function. This can lead to very difficult-to-read code. Another example which shows the difficulty of navigating and understanding C++ code is where we have a member function 'get-name()' attached to

object M. M -> get-name()

Given such a statement on is own, it is very difficult to figure out which class the method 'get-name()' is attached.The possibilities are that:

'get-name()' is a member function of class EMPLOYEE of which M is an instance.
'get-name()' is a member function of one of the ancestor classes of the EMPLOYEE class, if the subclass has not overridden the 'get-name()'method.
'get-name()' is a member function attached tosubclass of EMPLOYEE class, M is a pointer to an EMPLOYEE object and 'get-name' function isvirtual.

The importance of navigating and understanding code for maintenance purposes can not be overemphasized. The fact that navigation through C++ source code is a serious impediment to maintenance has also been conformed by Wiener [Wie94], Wybolt [Wyb90].

C++ programming environments

A programming environment is an integrated set ofinteractive tools that include at least a language interpreter, an editor, code browse, debugger and code finding systems. C++ offers no standard class libraries except the iostream library. There are plenty of third partyvendors that are now offering special purpose C++ libraries. The areas of application include graphical user interface, mathematics and data structure libraries.

Conclusion

In this paper we have presented our experiences from using the C++ programming language and an object-orented design methodology for the development of a prototype analysis tool for inheritance relationships.
We noted that inheritance, polymorphism and dynamic binding, reuse and strong typedness are very powerufl features of the C++ language. However we also noted that C++ is a language with the following properties:

a long learning curve,
very difficult to understand,
leaves the programmer to resolve most problems, and
needs standardization.

The fact that C++ leaves most of the decisions to the programmer and the fact that the C++ code is very difficult to understand suggest that there is need for more comprehensive tools to help developers understand the design decisions in developed systems. We hope that the lessons explained in this paper will not only enhance the understanding of the language, but also be very useful to anyone who intends using the language with any other object-oriented design methodology.

References

[Car92] T. Cargill. C++ Programming Style. Addison-Wesley,1992.

[Cop92] J. Coplien. Advanced C++ Programming Styles and Idioms. Addison-Wesley, 1992.

[Dum93] G. Dumpleton. OTC Software Environment Release 3.0, OTCLIB Manual. OTC Limited, Australia, 1993.

[JSS90] P. Jossman, J Shark, and E. Schiebel. Climbing the C++ Learning Curve. In Proceedings of the USENIX C++ Conference, pages 10-23, 1990.

[Len92] D. Lenkov. C++ Standardization Top issues. Journal of Object-Oriented Programming, June:67-71, 1992.

[Ous93] J.K Ousterhout. Tcl and the Tk Toolkit. Computer Science Division, Deaprtment of Electrical Engineering and Computer Sciences, University of California, Berkely, 1993.

[RW92] D. Reed and J. wyant. How safe is C++? Journal of Object-Oriented Programming, May:69-72, 1992.

[Str86] B. Stroustrup. The C++ programming Language. Addison-Wesley, 1986.

[WBWW90] R. Wirfs-Brock and B wilkerson, and L. Wainer. Designing Object-Oriented Software. Prentice Hall, 1990.

[Wei94] R. Wainer. A comparison of Object-oriented Languages. In Proceedings of the Object-Expo Europe Conference, pages 263-270, September 1994.

PARALLEL HIERARCHICAL ALGORITHM FOR IDENTIFICATION OF LARGE-SCALE INDUSTRIAL SYSTEMS

Boris Janković and Vladimir B. Bajić

Centre for Engineering Research, Technikon Natal,
P.O.Box 953, Durban 4000, Republic of South Africa
Tel.: (+27) 31-204-2560, Fax: (+27) 31-204-2560
e-mail: bajic.v@umfolozi.ntech.ac.za

Abstract

Parallel execution is the most powerful method of speeding up numerical analysis. However, numerical complexity increases much faster than the problem size. For this reason the computational effort required for the analysis of systems of big dimension, or generally speaking, large-scale systems, will be substantial even with application of parallel computer architectures. This leads to necessity for proper model reduction, as well as decomposition of the physical problem into a set of smaller-size problems. In this paper we propose a method which comprises both of these demands and results in an algorithm for parallel hierarchical identification of reduced-size large-scale models.

Introduction: Large-scale systems and related numerical problems

Large-scale systems (LSS), sometimes called complex systems, are usually defined as systems that consist of large number of interacting subsystems (*Si*ljak 1983). Apart from large dimensionality of such systems, the nature of interactions between the subsystems can be particularly complex (*Si*ljak 1983). Very often, there are difficulties even in identifying inputs and outputs of such systems (*Si*ljak 1983). Also, numerical difficulties in analysis of even relatively low-order MIMO systems may be significant, and since in many cases LSS can be treated as MIMO systems, the same type of numerical problems is immanent to them too. One of the most distinguishing characteristics of LSS is the fact that "one-shot" approach methods can not be, in general, successfully used in the study of their behavior. This poses a significant problem if an on-line analysis of such systems is needed. For instance, in the case of self-adaptive controllers for LSS that change behavior relatively fast compared to dominant time constants of the LSS, obtaining a satisfactory model within a given time constraint may be a very difficult problem which is sometimes impossible to solve.

One of the most common approaches for speeding up calculations, in general, is utilization of parallel processing algorithms. In this paper we investigate some possible benefits by parallelism at the appropriate hierarchical level that may be used for implementation of faster self-adaptive control of LSS. There are several levels of parallelism related to algorithm execution. What they all have in common is that a sequential problem is somehow transformed into an equivalent form that is suitable for parallel processing. Conceptually, the lowest hierarchical level is embedded on instruction level, and it is closely related to underlaying (multi)processor architecture and corresponding compiler design (Malinowski *et al.* 1985). The initial sequential problem is still very much independent of this parallelism and details of tasks necessary to convert such a problem into one suitable for parallel processing are often hidden from the application level. SIMD (Single Instruction, Multiple Data) and MIMD (Multiple Instruction Multiple Data) are examples of parallel architectures (Malinowski *et al.* 1985).

Partially overlapping with this one is the next hierarchical level in which sequential, mathematical (mostly numerical) algorithms are transformed into parallel ones. There are many examples of these including partial differential equations, vector addition etc. (Lei *et al.* 1985). Sometimes, specialized parallel processors are specifically build for efficient execution these tasks. However, it is obvious that such algorithms and processor realizations can not be used for other types of

numerical problems. At this point it is interesting to note that analog computers are capable of fast simulations because all of their elements work in parallel (Pearce 1985).

Still higher hierarchical level of parallelism is obtained at the application level. In this case the initial problem is broken into a set of smaller-size problems which can be then solved in some phases independently of each other. For such a situation problem decomposition depends crucially on the nature and structure of the actual problem. For example, principles used for decomposition in circuit analysis problems are not applicable directly to ecological systems or economic systems. However, once obtained via decomposition approach at this level, subproblems are fairly autonomous and they are very loosely dependent on the underlying processor architecture. This goes up to the degree that subproblems can be solved independently on separate computers.

This problem decomposition approach is often the only way of reducing the rapid increase of the size of numerical problems as the problem size increases. Parallelism offered by parallel computer architectures will, naturally, significantly decrease the time necessary for solving the numerical problems, but even in the case of problem decomposition, the problem size can reach the level when this parallelism will not be fast enough to complete the task given within the prespecified time interval.

A very prominent class of systems that exhibits numeric difficulties are large-scale dynamic systems (*S*iljak 1983). Transformation of problems associated with LSS analysis and design into a form suitable for parallel processing is not easy. These problems are not only due to the so-called "curse of dimensionality", but also due to the incomplete knowledge of subsystem interactions. Another example of the increased numerical complexity could be large-scale optimization problems. The most effective techniques for breaking up the original problem into a set of smaller ones are decomposition-coordination (Schoeffler 1971) and decomposition-aggregation methods (*S*iljak 1983). The first one is very useful for hierarchical control, while the other is good for model reduction. In what follows we propose a technique that uses decomposition principle and model reduction in order to provide a significant parallelism in the identification of a reduced order models of LSS.

The results are illustrated by a simulation example.

Decomposition as problem size reduction technique

Following mainly Guardabassi (1982) let S represent a solution set of an abstract problem P. The problem P can be modelled (defined) as an ordered triple $P = (D, \pi, Z)$, where D is a data set (over which the problem is defined). Here, the mapping $\pi : D \ni d \rightarrow \pi(d) \in Z \ \ (Z = im\,\pi(D))$, is called an intrinsic mapping of the problem P. Therefore, finding a solution of a problem P for a given d means to find (any) element $s \in Z$, such that $s = \pi(d)$ and where $Z \neq \emptyset$. In this case we say that P has solution(s). If there is only one $s = \pi(d)$, $s \in Z$, $Z \neq \emptyset$, then we say that the problem P has a unique solution for d. In the case when $\forall d \in D$ there exist only one $\pi(d) \in Z$, the mapping π is called the intrinsic function of P. In the case $Z = \emptyset$, P has no solution for d. We will consider only the situation when π is a function.

When the problem $P = (D, \pi, Z)$ is a difficult (complex) one, we would like to transform it into some other (ideally equivalent) problem $P' = (D', \pi', Z')$, which is easier to solve. For the purpose of building our identification procedure, we will assume that the problem P' belongs to the class of composite problems. A composite problem of order N consists of N subproblems which may, or may not, be mutually dependent. Suppose that subproblem P_i of the composite problem P is defined by $P_i = (D_i, \pi_i, Z_i)$. From this, we see that it has its local data and local solution set, as well as its intrinsic function. To obtain the global solution of the composite problem P, it is necessary to consider the global data set D (which generally will not be union of local data sets i.e. $D \neq \bigcup_i D_i$), set of interaction functions which show dependencies of local data sets on other subsystems' local solutions, and finally, global solution function which relates global solution to local solutions.

A very useful tool for representing subsystem interactions, and generally the structure of LSS, is

graph theory. Each node can either represent a subproblem or its local data, and branches of the graph represent the subsystem interactions. If such a (di)graph is acyclic, then the local data for each of the subproblems P_k depends only on local solutions of subproblems P_i, $i = 1, 2, ..., k-1$, and possibly itself, but not on other local solutions, i.e. on solution of P_i, $i > k$. In this case all subproblems can be solved in a sequence. However, if each subproblem has its own, autonomous, local data, that are not influenced in any way by other subsystems, then parallel solvers (algorithms) can be used . This is the principle that we will utilize in our approach to hierarchical identification.

Parallel Identification

From previous considerations it is clear that decomposition principle can be taken as the basis for implementation of parallel processing. It is successfully used in many aspects of LSS analysis, such as simulation (Malinowski *et al.* 1986). One problem, however, in which decomposition principle can not be applied directly is identification. The problem is that output of identification procedure is a model, so, initially, there is no model to decompose. To be able to transform an identification problem into a form suitable for parallel processing, we have to approach this indirectly.

When dealing with dynamic systems, similar considerations may be applied, but this time instead of considering problem models we consider systems. Clearly, we may have global systems representation (which is some form of input-output representation that hides its internal structure), as well as a composite model representation, in which the internal structure (i.e. subsystems and their interactions) is preserved to some extent. The relation between the two representations is given in Takahara (1982). Only for some special cases this relation will be isomorphic; in reality global models may have numerous equivalent representations as composite models.

This, however, is important for us as we can assume that model resulting from identification process will have its equivalent composite representation. Suppose that the result of identification of some given input/output data is a model M in global model representation. Based initially on Bajić (1995), it is argued in Janković (1996) that model M can be approximated by its composite representation M_{cm} in such a way that, if this representation is taken in a certain form, then a two-phase algorithm results: in the first phase some composite model components are identified in parallel, and in the second phase subsystem interactions are determined. The problem of matching the global model with the one in the composite representation has its counterpart in control theory where it is called "exact model matching procedure" (Moore and Silverman 1972). Our goal is to match some fictious, "assumed to be true" model M with a composite one using output feedback. This procedure due to way how it is implemented belongs to hierarchical type of identification. Analysis of the ability of such approach to approximate the global model can be found in Janković (1996). The particular composite forms for identification of SISO processes are given in Janković and Bajić (1996) and for MIMO systems in Bajić and Janković (1997).

In this paper we will combine the model reduction and parallel hierarchical identification to speed up the modeling necessary for self-adaptive controllers of LSS. With regard to model reduction, we take approach opposite to the one utilized by Obinata and Inooka (1976), Ouyang *et al.* (1897), who select system modes according to their contribution to power spectrum of output. In our case since there is no model initially, we use the method to discard the portion of power spectrum that has small contribution to total power. After reducing the bandwidth of such systems we can apply parallel identification procedure more efficiently to come to the subsystem models, and finally to the composite model.

LSS model description for parallel identification

In this section we give the mathematical description of the assumed LSS model to which the proposed parallel identification procedure will be applied. Let us assume that the original system, which is a SISO one with the input signal u and the output signal y, is composed of N subsystems S_j which mutually interact via the interconnection subsystem S_I. We assume that each of the subsystems S_j is described by the transfer function G_j and a dead-time L_j. Thus for the j-th subsystem S_j we

have the operator equation

$$\begin{aligned} y_j &= G_j(s)e^{-L_j s}u_j = \frac{B_j(s)}{A_j(s)}e^{-L_j s}u_j \\ &= \frac{b_{mj,j}s^{mj} + b_{mj-1,j}s^{mj-1} + \ldots + b_{1,j}s + b_{0,j}}{s^{nj} + a_{nj-1,j}s^{nj-1} + \ldots + a_{1,j}s + a_{0,j}}e^{-L_j s}u_j \end{aligned} \tag{1}$$

where y_j and u_j represent the output and the input signals of S_j, respectively. We assume the original system will have reasonably good representation as a LSS of the form

$$y_j = G_j(s)e^{-L_j s}u_j, \;\; j = 1, 2, ..., N \tag{2a}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ ... \\ y_N \end{bmatrix} \tag{2b}$$

$$y = \left(\mathbf{c}_{cm}e^{-Ls}\right)\mathbf{y} \tag{2c}$$

$$u_j = u + m_j, \;\; j = 1, 2, ..., N \tag{2d}$$

$$\mathbf{m} = \begin{bmatrix} m_1 \\ ... \\ m_N \end{bmatrix} \tag{2e}$$

$$\mathbf{m} = \mathbf{A}_{cm}\mathbf{y} \tag{2f}$$

where $\mathbf{y}$ is the vector whose components are the outputs of subsystems, $\mathbf{m}$ is the vector of feedback signals, $\mathbf{A}_{cm}$ is the $N \times N$ feedback matrix, $\mathbf{c}_{cm}$ is the N-component vector combining the subsystem outputs, and L is the dead-time in the LSS model (1-2) separate from those included in the individual subsystems S_j.

The purpose of the identification is to determine the coefficients in the transfer functions G_j, $j = 1, 2, ..., N$ and dead-times L_j, $j = 1, 2, ..., N$, for subsystems S_j, as well as the matrix $\mathbf{A}_{cm}$, the vector $\mathbf{c}_{cm}$ and the dead-time L. The proposed parallel identification procedure by reduced-order models can be stated as follows. In the first step, the bandwidth of the output signal y is reduced by discarding those frequencies with insignificant power contribution. Secondly, transfer function models $G_j(s)e^{-L_j s}$ of orders n_1, $n_1+1, ..., n_1+N-1$ are identified. This can be done in parallel. In the final step, parameters $\mathbf{A}_{cm}$, $\mathbf{c}_{cm}$ and L of the composite model (1-2) are identified.

Complexity analysis

In this section we give a rough estimate of the complexity of the identification procedure proposed. We would like to analyze possible benefits of the proposed decomposition and identification method. In order to do this we must somehow relate the computational effort with the problem size. Let us assume that in the parameter estimation problem, the problem size is defined as the number of parameters entering the optimization procedure. For the transfer function models

$$y = G(s)e^{-Ls}u = \frac{B(s)}{A(s)}e^{-Ls}u = \frac{b_m s^m + b_{m-1}s^{m-1} + \ldots + b_1 s + b_0}{s^n + a_{n-1}s^{n-1} + \ldots + a_1 s + a_0}e^{-Ls}u \tag{3}$$

the number of parameters that need to be determined is

$$\Theta(G, n) \leq 2n + 1$$

since $m \leq n$ due to physical realizability. Thus in the worst case one gets

$$\Theta(G, n) = 2n + 1$$

This relation is linear. For the composite (LSS) model (1-2) we have

$$\Theta(CM, n) = \sum_{i=1}^{N}\Theta(G_i, n_i) + \Theta(IM, N)$$

where $\Theta(CM, n)$ is the number of parameters for composite model of order n, $\Theta(G_i, n_i)$ is the number of parameters for each of the subsystem's transfer function, where n_i is the order of $i-th$ subsystem, and $\Theta(IM, N)$ is the number of parameters for the interaction subsystem S_I.

To get the expression for the maximal number of parameters of the composite model (1-2) that needs to be determined simultaneously we note that, without loss of generality, for the numbers of parameters of subsystems S_j, $j = 1, 2, ..., N$, the following holds

$$\Theta(G_1, n_1) \leq \Theta(G_2, n_2) \leq \ldots \leq \Theta(G_N, n_N)$$

as subsystems S_j can be numbered in such a way. Since all parameters of a subsystem S_j can be determined independently of the parameter identification procedures for S_i, $i \neq j$, then the number of parameters $\Theta(G_N, n_N)$ determines in a way the maximal computational effort and time needed in any branch of sequential processing for obtaining parameter estimates for any of the systems S_j in parallel computation. For the interaction subsystem S_I, if subsystem's dead-times are fixed, we can write

$$\Theta(IM, N) = N^2 + N + 1$$

where N^2 term comes from the matrix $\mathbf{A}_{cm}$, term N comes from vector $\mathbf{c}_{cm}$, and the last parameter is the composite model dead-time. It should be noted that the composite model (1-2) is of the order $n = \sum_{i=1}^{N} n_i$.

For the first step in parallel processing identification, the maximal number of parameters to be determined in an 'one-shoot' fashion is $2n_N + 1$, as already asserted. For the second phase the number of parameters is $N^2 + N + 1$.

Let us assume that the number of instructions necessary for optimization of the n-parameter problem is proportional to the square of number of parameters, i.e. proportional to n^2. This is a very conservative assumption and in any real-world situation this ratio is much larger. However, even with this we will show a great advantage of the method proposed. So the total number of instruction needed for the largest sequential processing demand in any of the parallel processing branches will be

$$\begin{aligned} ins &= \Theta(G_N, n_N)^2 + \Theta(IM, N)^2 = (2n_N + 1)^2 + \left(N^2 + N + 1\right)^2 \\ &= (2n_1 + 2N - 1)^2 + \left(N^2 + N + 1\right)^2 \end{aligned} \tag{4}$$

For the composite model given as (3) in the 'one-shoot' approach the number of parameters that we need to determine simultaneously is

$$\begin{aligned} \Theta(G_{LS}, n) &= 2n + 1 = 2\sum_{i=1}^{N} n_i + 1 = 2\sum_{i=1}^{N}(n_1 + i - 1) + 1 \\ &= N^2 + (2n_1 - 1)N + 1 \end{aligned}$$

and the corresponding number of instructions will be

$$ins = \Theta(G_{LS}, n)^2 = \left[N^2 + (2n_1 - 1)N + 1\right]^2 \tag{5}$$

To have a measure of the relative complexity in the case of parallel identification approach and the 'one-shoot' approach we now form a ratio Φ between the number of instructions (4) and the number of instructions (5), and we call this the relative complexity index. This ratio takes into account the effect of decomposition. Thus we have

$$\Phi(N, n_1) = \frac{(2n_1 + 2N - 1)^2 + (N^2 + N + 1)^2}{[N^2 + (2n_1 - 1)N + 1]^2}$$

We see that Φ is function of the number of subsystems N and the lowest order n_1 of the subsystem models.

The comparison of relative complexity of the identification methods for the parallel processing identification and 'one-shoot' identification for the model (3) of the same model orders is given in Fig.1. Fifteen curves are shown, obtained for $n_1 = 2, 4, ..., 30$, and for $N = 2, 3, ..., 20$. The curve at the top is obtained for $n_1 = 2$, and the position of curves gradually goes down with the increase of n_1. The greatest reduction in the algorithm complexity is obtained for $n_1 = 30$ and for $N = 8$. This roughly corresponds to the 30 times reduced algorithm complexity in parallel identification method compared to the normal 'one-shoot' approach. Essentially, graphs in Fig.1 show that the parallel identification technique proposed makes a significant reduction in the required instructions during the identification (and the time required for that) compared to the 'one-shoot' approach..

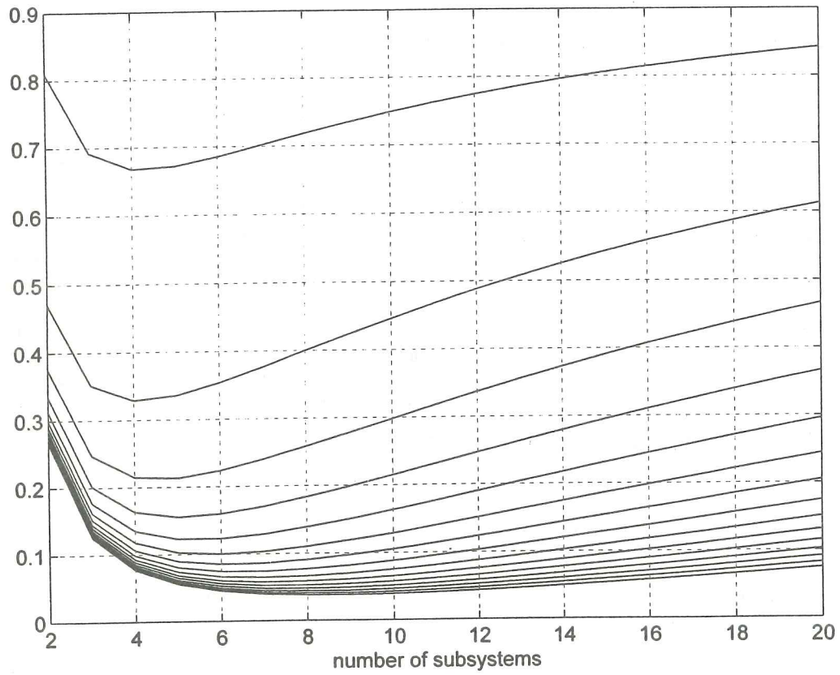

Fig. 1 Relative complexity curves

Example

The application of the method proposed is tested on the 3-channel autopilot model of order 74 (Simulink 1995). The input-output sequence is generated and a high frequency output components were truncated by passing the output signal y through a Butterworth filter. The final results of identification are shown in Figs. 2 and 3. As can be expected, the 'steady state' part of response is modelled more accurately because steady state is unaffected by low-pass filtering. This is clearly visible from Fig. 3 where the error between the response generated by the original model and the

response of the identified reduced order model is shown.

In the first phase three subsystem models of orders 2, 3 and 4 are identified. Note that identification of these three subsystems can be done in parallel. Subsystem dead-times were kept fixed to zero. The identified models in the first phase are as follows:

$$G_1 = \frac{1.1868s + 1.1189}{s^2 + 0.9201s + 1.1189} e^{-L_1 s}, \quad L_1 = 0$$

$$G_2 = \frac{3.6471s^2 + 8.6297s + 6.9719}{s^3 + 7.4694s^2 + 6.0564s + 6.9719} e^{-L_2 s}, \quad L_2 = 0$$

$$G_3 = \frac{0.8302s^3 + 8.2375s^2 + 32.6406s + 30.0417}{s^4 + 2.9463s^3 + 29.905s^2 + 24.2209s + 29.9163} e^{-L_3 s}, \; L_3 = 0$$

In the second phase the identified subsystems are coupled via the interconnection subsystem S_I to form the LSS model of the form (1-2). The parameters obtained in this phase are

$$\mathbf{A}_{cm} = \begin{bmatrix} 0.0078 & -0.3389 & -0.2879 \\ -0.0861 & -0.0457 & 0.0383 \\ 0.0334 & -0.1791 & 0.3577 \end{bmatrix}$$

$$\mathbf{C}_{cm} = \begin{bmatrix} 0.4234 & -0.3325 & 0.9200 \end{bmatrix}$$

$$L = 0.0031$$

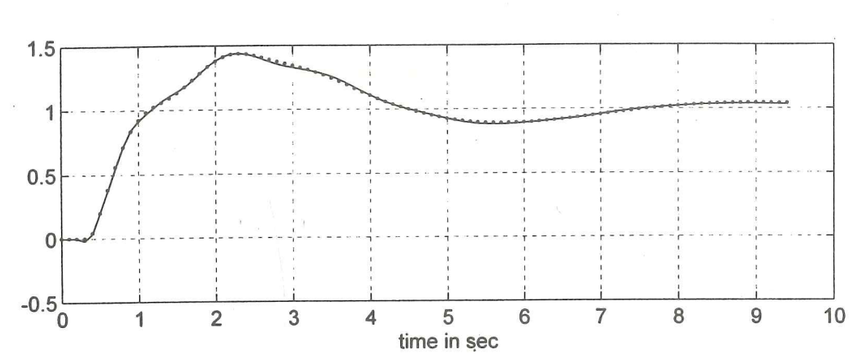

Fig. 2 Response of original model of the order 74 (solid) and reduced order model (dotted curve)

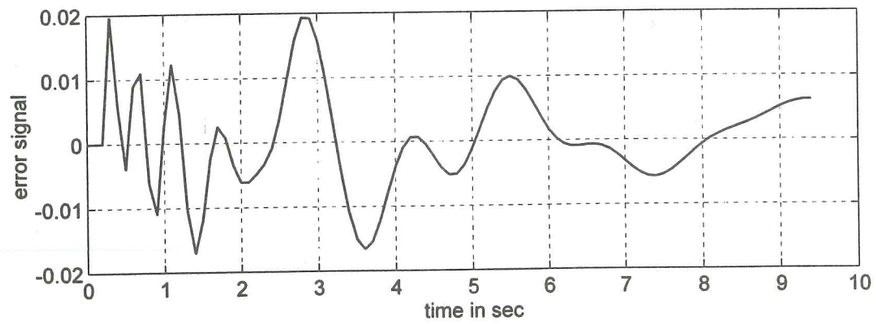

Fig. 3 Error of the response of the original model and reduced order model

Conclusions

A hierarchical method suitable for parallel identification of LSS is proposed. The method has advantages regarding computational complexity compared to the 'one-shoot' identification approach. Potential domain of application of this method is in the identification of complex industrial processes from real-time measurements. The method is presented in a form suitable for SISO systems, although it can be adapted to cater for MIMO system applications.

References

G.Authie and D. El Baz (1986). "A Multimicroprocessor for Parallel Processing", in "*Parallel Processing Techniques for Simulation*" (ed. M. A. Singh, A. Y. Allidina, B. K. Daniels), Plenum Press, pp. 229-238.

V.B.Bajic (1995). Private communication, January, RSA.

V.B.Bajić & B.Janković (1997). "Hierarchical identification of large-scale process plants", accepted for IFAC-IFIP-IMACS Conference, Belfort, France, May 20-22, 1997.

D.P.Bertsekas and J.N.Tsitsiklis (1989). "*Parallel and Distributed Computation - Numerical Methods*", Prentice-Hall.

G.Guardabassi (1982). "Composite Problem Analysis", *Large-Scale Systems*, Vol.3, pp.1-11.

B.Janković (1996) "*Hierarchical Identification of Large-Scale Systems*", Doctoral thesis, Technikon Natal, submitted.

B.Janković and V.B.Bajić (1996). "Reduced Optimization Complexity in Identification of Large-Scale SISO Processes", CSS '96, Brno, Czech Republic (to be published).

S.Lei, A.Y.Allidina and K.Malinowski (1985). "Clustering technique for rearranging ODE systems", *Proceedings of the First European Workshop on Parallel Processing Techniques for Simulation*, October, pp.31-44.

K.Malinowski, A.Y.Allidina and M.G.Singh (1986). "Decomposition-Coordination Techniques for Parallel Simulation", in "*Parallel Processing Techniques for Simulation*" (ed. M. A. Singh, A. Y. Allidina, B. K. Daniels), Plenum Press, pp.1-11.

K.Malinowski, A.Y.Allidina, M.G.Singh and W.D.Crorkin (1985), "Decomposition - coordination technique for parallel simulation, Part 1", *Large-Scale Systems*, Vol.9, pp.101-115.

B.C.Moore and L.M.Silverman (1972), "Model matching by state feedback and dynamic compensation", *IEEE Transactions on Automatic Control*, Vol.17, pp.491-496.

G.Obinata and H.Inooka (1976). "A Method for Modeling Linear Time-Invariant Systems by Linear Systems of Low Order", *IEEE Transactions on Automatic Control*, Vol.21, pp.602-603.

M.Ouyang, C.M.Liaw and C.T.Pan (1987). "Model Reduction by Power Decomposition and Frequency Response Matching", *IEEE Transactions on Automatic Control*, Vol.32, pp.59-62.

J.G.Pearse, P.Holliday and J.O.Gray (1985), "Survey of parallel processing in simulation", *Proceedings of the First European Workshop on Parallel Processing Techniques for Simulation*, October, pp.183-202.

J.D.Schoeffler (1971), *Static Multilevel Systems*, in *Optimization Methods for Large-scale Systems*, edited by David A. Wismer, McGraw-Hill.

*SIMULINK User 's Guide (*1995), MATLAB, The MathWorks, Inc.

D.D.Siljak (1983). "Complex dynamic systems: dimensionality, structure and uncertainty", *Large-Scale Systems*, Vol.4, pp.279-294.

Y.Takahara (1982). "Decomposability conditions for general systems", *Large-Scale Systems*, Vol.3, pp.57-65.

A Cultural Perspective On IT/End user Relationships (A description of Research in Progress)

AC Leonard
Department of Informatics
University of Pretoria
South Africa
Pretoria
0002

Abstract

An IT/end user relationship is described as intriguing and complex and is based on a *physical* and *abstract* dimension. These two dimensions enable one to fully describe the holistic nature of such a relationship and encapsulate the important elements of a support-oriented organization, namely mutuality, belonging, and connection. (Pheysey (1993)). In this paper a perspective on the importance of a supportive culture for sound relationships is described.

The importance of a *knowledge base* for both the end user and the IT professional before entering a relationship is also emphasized. Research in this regard shows that end user involvement should be seen as part of a relationship which exist between IT and its end users and which emerges during the life cycle of a software product.

Introduction

In order for the IT department to provide a proper service and support to its end users (customers), it is important that IT managers must first understand the meaning of "proper service" and how it should be managed. According to Luevano (1991), even if the IT department's reporting mechanisms may indicate that IT is meeting its objectives and doing an adequate job, these reports have no value if, in users' eyes, the IT department is not providing the expected service. He further states that the typical factors that contribute to negative perceptions on the end user side are, amongst others, a lack of communication between the IT department and the end users, and an IT department that is not "user driven".

The understanding and managing of these factors are very important. Global competition and challenges demand that the quality of services and support offered by IT should more than satisfy end users. In his article De Jarnett (1993) emphasizes the fact that everything IT does should be focused on fulfilling the needs of the customer and that the new goal should be to have delighted customers.

Di Carlo (1989) states that there is wide recognition for the maintanance of better sevice levels and that this should be a major strategic business objective.

According to Carey (1988) this involvement increased the likelihood that the resulting system will be satisfactory, as well as the amount of time necessary to complete the development task. Furthermore, it normally ensures that the amount of time and money spent on after service activities (maintenance) is greatly reduced. It is therefore of the utmost importance not only to ensure that the IT department knows how to manage end user involvement, but also that end users know why they should get involved and what role they need to play. In this regard, Ives et al (1984) also mention that the most commonly used indicator of success in user involvement studies has been user information satisfaction.

If one looks at what De Jarnett (1993) has to say in his article about this whole issue, it is easy to realise that not much has happened in this critical area. The question is why not? One could also ask whether all the complexities of the whole picture are presented?

In this paper some important issues regarding relationships[1] between IT and its end users are addressed. The complexities of such relationships will be explained as well as how it should be managed. Furthermore, I will explain why a supportive culture is important for sound relationships and which elements are important in order to ensure that such a culture will prevail.

In section 2 the diversity of end user involvement situations are briefly explained whereas in the third section the concept of an IT/end user relationship is defined and described as the umbrella for all the "contact" which takes place between IT and its end users. Thereafter the factors are described which will contribute to the creation and enhancement of a supportive culture. The importance of a 'paradigmatic break' from the prevailing orthodoxy in the management of IT/end user relationships is emphasized in the conclusion.

End user involvement

A few years ago, user contact was mainly focused on user involvement during systems development in a mainframe environment. Nowadays, with all the recent developments in technology, this contact varies in nature. For example, the growth in end user computing (EUC), which started in the 1980s, implies a new way of "contact" the IT environment has to deal with. According to Cotterman *et al* (1989) this significant phenomenon is ranked among the 10 most important MIS management issues for the 1980s.

[1] The term IT/End user relationship is defined in a separate paragraph in this paper.

As far as the mainframe environment is concerned, three different stages of IT/end user contact can be identified, namely:

* The request stage, during which the user communicates his/her needs in order to activate the IT department to develop a new product or to give the necessary support. (The pre-development phase).

* During development, which entails user involvement at different levels depending on the development strategy that is followed and

* During the production phase or post implementation phase - in other words, after the product has been placed in operation.

During all these different stages the end user and IT professionals are involved in different formal and informal social communication structures. The most common of these structures are project meetings, end user groups, training and JAD sessions[1].
According to Carey, end user involvement is an often overlooked aspect in developing easy-to-use systems which consist of the methods used to get users directly involved in the design, evelopment, testing, implementation and maintenance of the system. He states that end users are involved in different ways after an information system has been implemented, of which the following are the most important:

* End user training;
* User resistance to change;
* User feedback for system evaluation; and
* User acceptance of the system.

Of course, the abovementioned "ways of involvement" are different social communication structures in which the end user participates. In general, the importance of these social communication structures cannot be over emphasized, because it has a direct influence and impact on the quality of the end product, but more specifically on the efficiency of use by the end users. As Carey puts it: "All of these issues impact on the success of the information system."

Furthermore, as we all know, nothing is more pleasant than an end user who is positive about a software project and who gives all the necessary cooperation during the life cycle of the product. According to Hawk (1992) an important and widely recognised goal of getting end users involved, is to obtain a positive change in user attitudes and perceptions. In their articles Ingersheim (1976) Baroudi (1986) and Olson *et al* (1986) again point out that in order to improve user attitudes towards the resulting system, they should get involved.

In the article of Roode *et al* (1989), reference is also made to user involvement during systems development and the concept of "user

[1] The existence of social structures is a characteristic of a relationship and is discussed in more detail in the following section.

involvement transactions" is introduced. According to them, user involvement during systems development has two overall objectives, namely to achieve improved systems quality and to increase user acceptance. This of course also contributes to the eventual success of an information system. As far as involvement is concerned, it is defined to ".. cover generically all activities which a person who is involved may perform relative to the systems development project being undertaken..". The generic term 'user involvement' includes a number of 'user involvement transactions' which are observable units of involvement.

Characteristics and definition of an IT/End user Relationship

First of all a definition of such relationships are given in order to eventually show the important role and influence that organization culture may have on these relationships. Furthermore, it is important to take cognisance of the fact that the success of information systems depends to a great extent on the soundness of these relationships.

The abovementioned different "ways" in which end users are involved before, during and after the implementation of an information system are embodied in so-called "relationships". When an end user initially states his need(s) for a given business environment, a *new relationship* grows or develops therefrom. This means that whenever the IT department starts with a new project, a new relationship is created with the end user. In this regard Dahlbom et al (1993) states that a systems development process is actually an intervention into current business processes. As he puts it:

"Intervention is an approach to systems development with and by the users. Responsibilities are negotiated and shared between systems developers and users".

The nature of a relationship is therefore not only dependent on the business environment that will be involved during the whole systems development life cycle but also on the systems development approach that is going to be followed[1].

According to Carey (supra) there are basically three partners during the life cylce of a software product, namely the IT professional(s), the end user and the product itself. "The three participants in this relationship ...form an intriguing and complex relationship." (Carey (*Op. cit.*))

A definition of a relationship between IT and the end user during the life cycle of a software product will have to consist of two dimensions, namely a *physical dimension* and an *abstract dimension*. The physical dimension should describe those elements which are necessary in order to enable contact between IT and its end users, whereas the abstract dimension should describe the soft issues of a relationship. These two

[1]The hard, soft and dialectic approaches as described by Dahlbom et al refer.

dimensions enable one to fully describe the holistic[1] nature of such a relationship and encapsulate the important elements of a support-oriented organization, namely mutuality, belonging, and connection as mentioned by Pheysey (1993) in his book *Organizational Cultures*[2].

As far as the *physical dimension* is concerned, the following elements could be seen as the most important:

* People[3]

  As far as people are concerned, a relationship consist of all the responsible people who are involved in the systems development life cycle at a given time. "Responsibilities are negotiated and shared between systems developers and users" (Dahlbom (*op. cit*))

* Technology

  Technology may be seen as one of the most important elements in such a relationship, which enables the people who participate in the relationship to communicate with one another. Apart from the normal communication technology[4], facilities like *help desks* and *internet* are of the most important role players in this regard.

* Procedures

  In this regard one can refer to two types of procedures, namely organizational procedures (like standards and policies) which already *exist* and which can be seen as a given and *new* procedures that are being created by people because of their interaction with the given procedures and technology. In this regard DeSanctis (1994, p. 125) states:

> "Prior to development...structures are found in institusions such as reporting hierarchies, organizational knowledge, and standard operating procedures...the structures may be reproduced so as to mimic their nontechnology counterparts, or they may be modified, enhanced, or combined with manual procedures, thus creating new structures within the technology."

[1]Under the description of the abstract elements the term 'holistic' is described in broader terms.

[2]The nature of a support culture is discussed in more detail in the following section.

[3]As human beings, people are viewed in this regard as the *physical enablers* who initiate, create, participate and maintain relationships because of their interaction with one another in the IT/End user environment. This is inline with the adaptive structuration theory of DeSanctis (1994) which is used for theory building in my research; see also Orlikovski (1992) who's structurational model of IT consists of three components, viz, human agents, IT, and institutional properties of organizations.

[4] Telephones, fax machines and cellular networks.

* Structures

Depending upon the "type" of end user and therefore the service and support that will be offered, relationships will differ in content as far as *formal and informal* social communication structures are concerned. The most common of these structures are project meetings, JAD sessions and end user group meetings.

For example, end users in the pure EUC environment would have much less contact with the IT department than users who are totally dependent on IT for their software products. In the first example, relationships will probably consist of some informal social structures in order to give the support this type of end user will need from time to time. In the second example, relationships will probably consist of formal social structures like scheduled end user group meetings, in order to give the end user the necessary support and service.

These structures are mainly used by the end user to inform IT about the business needs, but also to ensure that the emerging product stays on track. On the other hand, it is used by IT to get hold of business needs but also to get clarity on already stated needs. Furthermore, these communication structures serve as inquiry forums for both the IT professional as well as the end user during which they learn and understand the new environment of the emerging system or that part of the organization they are actually redesigning. According to Dahlbom (1993, p 122) designing a computer system is really a means of redesigning the organization. According to him the challenge is to understand and change established traditions in the user organisation as well as in the project group.

As far as the *abstract dimension* is concerned, the following characteristics are the most important:

* They are dynamic

The nature of the relationships between the IT department and its end users will, *inter alia*, depend upon the "type" of end user, as well as upon regarding the end user as a human being. According to Umbaugh (1991, p. 140), when talking to end users the IT professional should always keep in mind the users' concerns, problems, environment, and responsibilities in terms of opportunities for IT services and support. Furthermore, he says, continuous contact with end users gives IT the opportunity to gain more insight into their problems. This, of course, demands that relationships should be adaptive to changing circumstances.

* They are sensitive to change[1]

One of the most difficult problems during systems development is to cope with users' resistance to change. Because of the social nature of relationships, any form of change which is initiated on the IT side will disturb such relationships.

* They have a knowledge base

According to Carey (*supra)* at least part of the design problem seems to be that the analyst/designer is working from his or her own perceptions of the user's needs which often include unrealistic expectations of user *knowledge* and an often mistaken idea of user desires and requirements. Furthermore, he says, the analyst views the computer from an expert's point of view and an often technical perspective. The user views the computer as a potentially useful tool, but from a more general orientation. These two views are quite different and are often incompatible and in conflict.

The resolution of these different perspectives is possible only when the analyst begins to understand the needs, requirements, and desires of the user in order to design and produce the system properly[2]. Users must also aid in this resolution by understanding the limitations of the computer systems they desire as well as by developing a thorough and specific understanding of their own needs.

The abovementioned explanation of the complex world of perceptions, attitudes and approaches towards developing software products by IT professionals for the end user, forces us to a point where one can say that in order to overcome the most serious problems during this communication process in a relationship, a knowledge base[3] of some kind is necessary before entering a relationship.

* They have a supportive culture[4]

In order for a relationship to be sound continuous support and mutual understanding, inter alia, need to be elements of such a relationship. According to Pheysey (*op.cit.*), a support-oriented organization have the elements of mutuality, belonging, and connection. Furthermore, an appreciative form of control should be applied, which means: "management is seen to be a process focused

[1]Research in this regard still in progress.

[2]This problem also relates to Brunwik's law of the lens which is discussed in the next section.

[3]Research in this regard is described in the next section.

[4]The importance of a supportive culture for maintaining sound relationships is discussed more fully in the next section.

om maintaining balance in a field of relationships" Gadalla and Cooper (1978) quoted by Pheysey (*op.cit.)*.

* A co-operative behaviour pattern is followed by the participants

Co-operation is not a fixed pattern of behaviour, but is a changing, adaptive process directed to future results. The representation and understanding of intent by every party is therefore essential to co-operation, and so the role of communication in co-operation is important. Clarke *et al* (1993).

Furthermore, they say, co-operation can also create new motives, attitudes, values and capabilities in the co-operating parties and therefore such behaviour will help to create and maintain a supportive culture.

* They have an holistic nature

The important elements which make up a relationship between the IT department and its end user at a given time are organized together as a whole. If any of these elements are disturbed in a negative sense, the whole relationship between the IT department and its end users is undermined.

In order to ensure proper end user involvement, it is of the utmost importance that the end user should understand its role and function as end user.

One of the typical characteristics of end-user behaviour is the lack of certainty of what they want which normally starts to disappear the moment they get exposed to different possibilities shown by an IT professional.

It is the responsibility of the IT department to create a culture with the necessary characteristics to ensure that the end user can fulfil its role. In such an environment the end user should feel comfortable, secure, knowledgeable, and useful. Furthermore, such an environment should incorporate the abovementioned factors that will not only ensure end user participation but will maintain a sound relationship during end user involvement. The question now is how can an IT department create and maintain such an organizational culture?

A supportive culture for sound relationships

First of all it is important to start with a definition of what is meant with the term culture. According to Towers (1992), an authoritative definition of a corporate culture is as follows:

"Culture is the commonly held and relatively stable beliefs, attitudes and values that exist within the organization"

Williams *et al* (1989).

Service and support are but two of the important issues that IT should concentrate on in order to survive. In a recent study of end users done by Daryl *et al* (1994) it was found that only 2% of the interviewees do not use a computer at all. Furthermore, the study revealed that the majority of the companies in the study did have an information center (IC), but a significant percentage of the end users were less than pleased with support provided by the IC. This indicates that the problem is serious and that a supportive culture does not exist in the majority of cases.

In a study done by Orlikowski (1992), she points out the importance of culture and the understanding of various viewpoints held by managers, users and system designers in implementing new technology. According to her, the viewpoints are significantly different, therefore they define success very differently. Unless these various viewpoints are surfaced and explored, use of technology can fail miserably, with everyone blaming the product rather than the true culprits - the corporate culture or the various groups' conflicting perceptions.

In order to bridge the "gap" between the world of the end user and the IT professional, especially as far as new and/or unexperienced participants in a relationship are concerned, it is of the utmost importance that all participants in such a relationship should be on the same "level" of communication. The understanding and managing of all communication factors are very important.

Research conducted so far has shown with reasonable certainty that the following factors are very important in order to change the mind frames of all participants in a relationship in order to ensure sound communications and proper involvement within the relationship - this in itself will cause the creation of a new culture between IT and its end users. According to Pheysey, in a supportive culture "people will contribute out of a sense of commitment to a group or organization of which they feel themselves truly to be members, and in which they believe they have a personal stake." (Pheysey (op. cit.) It is therefore of the utmost importance that every participant should have a proper understanding and realisation of and commitment to these factors:

* Knowledge of the *physical* and *abstract* nature of the IT/End user relationships as previously discussed

* An understanding of the functions and responsibilities of the end user as well as the IT environment

  As previously discussed[1], Sprague et al (1993), states that the old categories of end users are no longer appropriate, because of the new level of pervasiveness taken on by information technology. It is therefore important for an end user to know the category or *end user type* he belongs to in order to know what his responsibilities are and what kind of service and support he can expect from the IT side.

[1]Research in this regard still in progress.

* Knowledge of the ethical[1] grounds a sound relationships should be based on

* An understanding of the effects of organizational change on a relationship[2]

* Brunwik's Law of the Lens.

  The value of Brunswik's lens (Pitt & Bromfield, 1994, p.5) lies in its emphasis of the fact that the accuracy of a person's judgement depends apon the extent to which his mind represents the environment it attempts to predict.

* Cognisance of the "holistic-dilemma"

  As discussed previously, if any of the elements that make up a relationship is disturbed in a negative sense, the whole relationship between the IT department and its end users is undermined. It is therefore very important for all role players to be sensitive and aware of this dilemma, in order to ensure sound relationships.

* Empowerment of the end user[3]

* Levels of end user involvement depending on the end user type[4]

* Approach to systems development

  When developing information systems, development teams should discuss and decide what course of systems thinking (paradigm) they are going to follow. This is important because everyone on the development team should at least work within the same broad framework of thinking in order to ensure a collaborative atmosphere. This is especially important from the end user's point of view. According to Dahlbom (1993) it should be realised that only by working closely with the users the development team will we be able to develop useful information systems.

* The delegation trap as explained by Tony Gunton (1990).

  Tony Gunton's 'delegation trap' refers to the bad habits of end users who easily hand over systems to sub-ordinate staff to operate as soon as the 'novelty wears off'. In practice we have seen numerous cases where end users participate in the development process and as soon as they get bored, they delegate this important task to a sub-ordinate which may not be the 'right' end user.

[1].Research in this regard still in progress.

[2]Research in this regard still in progress.

[3]Research in this regard still in progress.

[4].Research in this regard still in progress.

Conclusion

The question is, what is to be done by an IT department or for that matter the organization which the IT department belongs to, that will help to create and maintain such an environment where a supportive culture between IT and its end users will prevail?

So far research in this area shows the importance of a 'paradigmatic break' from the prevailing orthodoxy in the management of the participants in the abovementioned intriguing and complex relationships.

I believe that there are several approaches or actions that IT can follow in order to create such an environment. However, it is very important that all employees on the IT side should be part of such a program of action and everyone should work towards this new paradigm of thinking.

One of the biggest dangers regarding the maintenance of a a supportive culture in an IT department is that this culture can change because of different forms of internal/ external change without even knowing it. According to Pheysey *et al (supra*) one should take note of Greiner's model which shows contingencies over time. Greiner called his model 'evolution and revolution as organizations grow'. The model is used by Pheysey to show a cyclic movement of control, starting in the supportive culture area, moving through power, achievement and role cultures and back to support culture. During this movement the organization must resolve, as they grow, the four crises down the centre. At this point, the question is: "Will the old crises recur, or will there be new crises to face?" (Pheysey (*op. cit.*))

Possible solutions which may help IT departments in creating such a supportive culture, might be to rethink the way their support is structured and managed. In his article, Scannell (1994) states that research among IT departments indicates that most need to adopt a new method of end user support. According to him many things need to be done of which restructuring the support function in order to implement all efforts effectively is necessary. This is sanctioned by Fischer (1994) who says "a well-designed IT support organization provides direct end user support and training, innovations to the working environment, consulting, and the necessary support mangagement."

As a conclusion a few practical suggestions are made in this regard.

- According to Fryer (1994), as companies become less hierarchical and employees are expected to shoulder more responsibility, operations staffers can be placed first in line for critical customer and end user support positions. This implies a much more direct IT/end user relationship.