Proceedings 1^{st} Student Colloquim 2003-2004
Computing Science
University of Groningen

Rein Smedinga Jan Terlouw

19 and 20 January

ISBN: 90-367-2126-1
Publisher: Bibliotheek der R.U.
Title: Proceedings 1^{st} Student Colloquim 2003-2004
Computing Science, University of Groningen
NUR-code: 980

Introduction

StudColl is a course that master students in computing science follow in the first year of their master study at the University of Groningen.

In the academic year 2003-2004 StudColl was organized for the first time as a conference. Students wrote a paper, participated in the review process, gave a presentation and were session chair during the conference.

The organizers Rein Smedinga and Jan Terlouw would like to thank all colleagues, who cooperated in this StudColl by collecting sets of papers to be used by the students and by being expert reviewers during the review process. They would also like to thank Angeniet Kam and Yvette Meinema from the Faculty of Arts for their help in organising this course.

In these proceedings all accepted papers are published. One chapter is devoted to organizational matters, list of participants et cetera.

Rein Smedinga
Jan Terlouw

Organizational matters

StudColl 2004 was organized as follows. Students were expected to work in teams, consisting of two persons. Angeniet Kam and Yvette Meinema of the Faculty of Arts gave an introductory lecture about general aspects of presentation techniques. After this, the student teams could choose between 15 sets of papers, that were made available through *Nestor*, the digital learning environment of the university. Each set of papers consisted of three papers about the same subject (within Computing Science). Soms sets of papers contained conflicting meanings. Students were instructed to write a survey paper about this subject including the different approaches in the given papers. The paper should also include ideas and conclusions about the subject students had developed themselves.

After submission of the papers individual students were assigned one paper to review using a standard review form (see Appendix A). The colleagues who had provided the set of papers were also asked to fill in such a form. Thus, each paper was reviewed three times. Each review form was made available to the authors of the paper through *Nestor*.

All papers could be rewritten and resubmitted, independent of the conclusions from the review. After resubmission each reviewer was asked to rereview the same paper and to conclude whether the paper had improved. Rereviewers could accept or reject a paper. All accepted papers can be found in these proceedings. All students were asked to present their paper at the conference.

Students were graded both on the writing process, the review process and the presentation. Writing and rewriting counted for 50% (here we used the grades given by the reviewers and the rereviewers), the review process itself for 15% and the presentation for 35%.

On January 19th and 20th, the actual conference took place. Of each writing team one author presented the results on Monday, the other presented the results on Tuesday. Both days, we had seven presentations. Six papers were accepted to be published in these proceedings.

Day 1: monday January 19th, 2004

List of participants

Eldering, B.
Es, E. van der
Neeteson, D.P.
Maneschijn, N.A.
Simonides, B.J.
Slagter, R.
Westerhof, A.C.

Program

9:00-9:10	opening
9:10-9:50	Simonides, B.J. *The many definitions of computability* **chair:** Neeteson, D.P.
9:50-10:30	Slagter, R. *Making LR Parsing Easier* **chair:** Maneschijn, N.A.
10:30-11:00	coffee break
11:00-11:40	Westerhof, A.C. *A survey of Bridging the Gab Between SE and HCI* **chair:** Eldering, B.
11:40-12:20	Maneschijn, N.A. *An overview of several noise removal methods* **chair:** Westerhof, A.C.
12:20-13:20	lunch break
13:20-14:00	Es, E. van der *Introduction to various color segmentation methods and their applications* **chair:** Simonides, B.J.
14:00-14:40	Neeteson, D.P. *ERP: Does it live up to its promises?* **chair:** Slagter, R.
14:40-15:20	paper is not accepted
15:20-15:30	closing

Day 2: tuesday January 20th, 2004

List of participants

Hofstra, M.
Kelder, M.
Noorlander, G.
Rossing, R.
Schotanus, A.
Starre, L.J. van der
Visser, W.T.

Program

9:00-9:10	opening
9:10-9:50	Schotanus, A. *ERP: Does it live up to its promises?* **chair:** Kelder, M.
9:50-10:30	Rossing, R. *The many definitions of computability* **chair:** Hofstra, M.
10:30-11:00	coffee break
11:00-11:40	Visser, W.T. *Introduction to various color segmentation methods and their applications* **chair:** Noorlander, G.
11:40-12:20	Starre, L.J. van der *Making LR Parsing Easier* **chair:** Schotanus, A.
12:20-13:20	lunch break
13:20-14:00	Noorlander, G. *An overview of several noise removal methods* **chair:** Visser, W.T.
14:00-14:40	Kelder, M. *A survey of Bridging the Gab Between SE and HCI* **chair:** Rossing, R.
14:40-15:20	paper is not accepted
15:20-15:30	closing

Contents

Making LR Parsing Easier

Laurens van der Starre, csg0017@wing.rug.nl
Robert Slagter, csg0057@wing.rug.nl
University of Groningen
The Netherlands

Submitted for the StudColl2004 conference

Keywords: LL-parsing, LR-parsing, top-down, bottom-up, recursive descent, push-down, sets-of-items, Pepper, 1NF, 2NF, 3NF, differences, alternative, easier, combining, comparison.

1 Abstract

The two most commonly used parsing techniques, LL and LR parsing, are generally considered to be entirely different approaches to parsing. In this paper we show that the methods are actually not that different, using a transformation of the underlying grammar. The disadvantages and advantages of this alternative approach are then discussed.

2 Introduction

LL and LR parsing are almost always described as two different parsing methods. LL parsing is often associated with terms like recursive descent, top-down approach and ease of construction. LR parsing on the other hand is more often found to be difficult to understand, and is associated with push-down automata and a bottom-up approach. In this paper we will show that in fact the "gap" between the two parsing methods is not that big, by creating a top-down approach (classically associated with LL parsing) for LR parsing. A number of methods for constructing such a parser have been devised (see for example [Pep99], [SB95], [Pij93], [Bea82]), but we will follow the approach found in [Pep99]. We will then evaluate whether parsing LR grammars using LL techniques is a better approach to LR parsing than the classical approach, using push-down automata.

3 Short recapitulation and definitions

The reader is assumed to have some basic knowledge on compiler construction and formal grammars, see for example [GBJL00]. We will shortly recapitulate some definitions we will need in this paper.

Definition 1 (Context-free Grammar) *A context-free grammar (CFG) is a four-tuple $(\mathcal{T}, \mathcal{N}, S, \mathcal{P})$, where $\mathcal{T}$ is the set of terminals, $\mathcal{N}$ the set of non-terminals, $S \in \mathcal{N}$ the start symbol and $\mathcal{P}$ the set of production rules $\mathcal{P} \subseteq \mathcal{N} \times (\mathcal{N} \cup \mathcal{T})^*$.*

Definition 2 (LL(k)-grammar) *Let G be a CFG, k is a natural number.*

$$S \overset{*}{\Longrightarrow}_{\text{lm}} uY\alpha \Longrightarrow_{\text{lm}} u\beta\alpha \overset{*}{\Longrightarrow}_{\text{lm}} ux \text{ and } S \overset{*}{\Longrightarrow}_{\text{lm}} uY\alpha \Longrightarrow_{\text{lm}} u\gamma\alpha \overset{*}{\Longrightarrow}_{\text{lm}} uy$$

k-level prefixes of x and y coincide. This implies $\beta = \gamma$. In words: the choice of the alternatives for the current non-terminal Y for the fixed left context u is uniquely determined by the first k symbols. This definition was taken and modified from [WM95].

Definition 3 (LR(k)-grammar) *Let G be a CFG, G is LR(k) from the conditions:*

1. $S \overset{*}{\Longrightarrow}_{\text{rm}} u_1 A_1 v_1 \Longrightarrow_{\text{rm}} u_1 x_1 v_1$
2. $S \overset{*}{\Longrightarrow}_{\text{rm}} u_2 A_2 v_2 \Longrightarrow_{\text{rm}} u_2 x_2 v_2 = u_1 x_1 v_3$
3. *k-level prefixes of v_1 and v_3 coincide.*

One can always deduce $u_1 = u_2$, $A_1 = A_2$, $x_1 = x_2$, $v_2 = v_3$. This definition was taken from [Tom].

In the remainder of this paper, we use capital characters (e.g. $S,\ N,\ Z_5$) as non-terminals and small characters (e.g. $a,\ b,\ k$) as terminals.

4 The classical view of LL and LR parsing

4.1 LL Parsing

LL-parsing is classically associated with a top-down approach and a recursive descent implementation. Parsers of this kind predict the input based on the production rules. This gives an almost "natural" conversion from a production rule to a procedure which implements the parser for that particular production rule. For example, consider the rule $A \to abC$. In a programming language, this will often be implemented as a combination of *matching* and *recursive calls*:

```
procedure procA();
  begin
    match('a');
    match('b');
    procC();
  end;
```

As one can see there is a clear mapping from (non)terminals to code fragments:

- Every terminal t maps to a call to `match(`t`)`.
- Every non-terminal N maps to a call to a procedure `procN`.

An LL-parsing of a grammar leads to a *pre-order* traversal of the parse tree, and grammars which have left-recursive production rules cannot be parsed by a LL-parser. If we look at a schematic view of the parse tree construction, we see that the tree is built up from the top, traversing down to the bottom, as shown in figure 1.

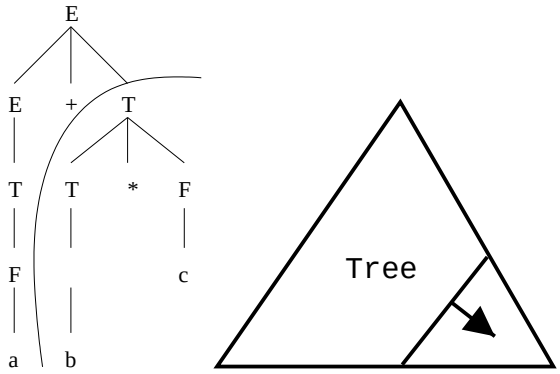


Figure 1: Parse tree

4.2 LR Parsing

LR parsing is much more powerful than LL parsing, as it also accepts left-recursive grammars. Unfortunately, creating a LR parser is more difficult, less "natural" than creating an LL parser. LR parsers do not predict the input based on production rules, instead they continuously search for non-terminals to which the input can be reduced. In order to do this, a *table-driven parser* is created, in combination with a stack. The table tells the parser which state to go to next, based upon the current state and the symbol on top of the stack. The stack on the other hand is used to push symbols and states on. The order in which a parse tree is traversed in LR parsing is *post-order*. Shown in figure 2 is a schematic view of a parse tree contruction. The tree is constructed from bottom to top.

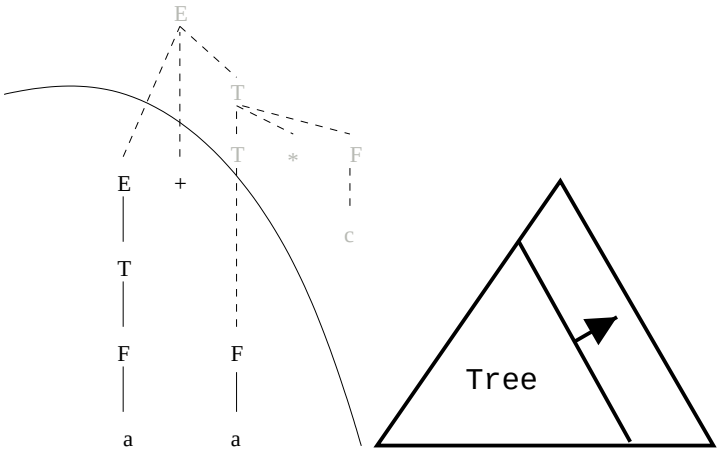


Figure 2: Parse tree

5 Combining LL and LR parsing

What we would like to have is a parser with the power of an LR parser, but which is constructed just as easy as an LL parser. Because LL and LR have for a long time been thought of as two different methods, this was never thought of as a possibility. But research in recent history has shown otherwise, and in fact there is a relationship between LL and LR parsing. The strategy is to add extra information to the LL parse tree or grammar and to obtain by using that strategy the LR parse order and power, while retaining the easiness of an LL implementation. There are a number of methods available but the one we will describe is explained in [Pep99]. The idea is to convert the grammar to a normal form which is better suited for parsing. The first step in this process is to add rule numbers, that is to add special *non-terminals* (i), $i \in \{1, 2, 3, \ldots\}$. Each alternative of each production rule gets a unique non-terminal (i). Note that these non-terminals *derive the empty string (ϵ)*, and are used only internally in the parser. Because these non-terminals derive ϵ, the language which the grammar generates, does not change compared to the original grammar. The grammar is then consecutively transformed to 1NF, 2NF and 3NF, again preserving equality between the grammars. The definitions are as follows:

Definition 4 (1NF) *A CFG is said to be in 1NF, if every production is of one of the three forms: $X \to YZ$, $X \to Y(i)$ or $X \to (i)$, with $X \in \mathcal{N} \cup \mathcal{Z}$, $Z \in \mathcal{Z}$, $Y \in \mathcal{N} \cup \mathcal{T}$. $\mathcal{Z}$ is a set of auxiliary non-terminals.*

Definition 5 (2NF) *A CFG is said to be in 2NF, if every production is of one of the two forms: $Z \to tz \wedge |z| \geq 1$ or $Z \to (i)z \wedge |z| \geq 0$, with $Z \in \mathcal{Z}, t \in \mathcal{T}, z \in (\mathcal{Z} \cup \mathcal{A})^*$. $\mathcal{A}$ is the set of action symbols (i).*

Definition 6 (3NF) *A CFG is said to be in 3NF, if it is in 2NF and there are no two productions $Z \to xu|xv$ the right-hand sides of which start with the same symbol.*

So in short: 1NF reduces the length of each alternative of each production rule to a maximum of two symbols, 2NF converts the original grammar to a grammer whose heads of the production rules consist only of auxiliary non-terminals (set $\mathcal{Z}$), and 3NF ensures that no two alternatives of the same production rule start with the same symbol. Different techniques can be applied to convert a grammar to 1NF, 2NF and 3NF (see for example [Pep99]), and we will use some of them in the next section, where we will compare a "standard" LR parser with a parser constructed using a 3NF-grammar.

6 Comparison

In this section we will construct an LR parser using both the classical and the alternative method as explained in section 5. We will then compare the two resulting parsers and come to the conclusion that the parsers are equivalent. The grammar we will use throughout the example is the following context-free grammar G:
$A \to aB$
$B \to b$
$B \to AC$
$C \to c$

6.1 Construction using the classical method

In order to create the parser, we calculate the states needed using the "Sets Of Items" method, which is (for example) described in [GBJL00]. The result of this algorithm is presented in table 1, and the corresponding ACTION/GOTO definitions in table 2. Note that we have added the production $S \rightarrow A \perp$, where the symbol $\perp$ indicates a succesful parse. The start symbol of the grammar is S. Fortunately we don't have any reduce/reduce or shift/reduce conflicts (so the grammar is LR(0)). We have kept the grammar this simple deliberately to make the example easy to understand.

State 0: Shift		**State 1: Accept**	**State 2: Shift**	
$S \rightarrow .A \perp$	*(Goto 1)*	$S \rightarrow A. \perp$	$A \rightarrow a.B$	*(Goto 3)*
$A \rightarrow .aB$	*(Goto 2)*		$B \rightarrow .b$	*(Goto 4)*
			$B \rightarrow .AC$	*(Goto 5)*
			$A \rightarrow .aB$	*(Goto 2)*
State 3: Reduce		**State 4: Reduce**	**State 5: Shift**	
$A \rightarrow aB.$		$B \rightarrow b.$	$B \rightarrow A.C$	*(Goto 6)*
			$C \rightarrow .c$	*(Goto 7)*
State 6: Reduce		**State 7: Reduce**		
$B \rightarrow AC.$		$C \rightarrow c.$		

Table 1: Sets of Items for the push-down automaton for G.

State	**a**	**b**	**c**	**A**	**B**	**C**	**Action**
0	2			1			Shift
1							Accept
2	2	4		5	3		Shift
3							Reduce using $A \rightarrow aB$
4							Reduce using $B \rightarrow b$
5			7			6	Shift
6							Reduce using $B \rightarrow AC$
7							Reduce using $C \rightarrow c$

Table 2: Combined ACTION/GOTO table for the push-down automaton for G.

6.2 Construction using the alternative method

Adding rule numbers

The first step is to add rule numbers to the production rules of G. Doing so yields:
$A \rightarrow aB①$
$B \rightarrow b②$
$B \rightarrow AC③$
$C \rightarrow c④$

Converting to 1NF

To ensure that the grammar conforms to 1NF, we make sure that the length of each alternative of each production rule is at most 2. We therefore introduce a number of auxiliary non-terminals Z_i:
$A \rightarrow aZ_1$
$B \rightarrow b②$
$B \rightarrow AZ_2$
$C \rightarrow c④$

$Z_0 \rightarrow A⓪$
$Z_1 \rightarrow B①$
$Z_2 \rightarrow C③$

Note that we added a ⓪ symbol and a corresponding production rule ($Z_0 \rightarrow A⓪$). The ⓪ symbol is used to indicate a succesful parse, just like we did in the classical construction using the $\perp$ symbol. The start symbol in this grammar is Z_0.

Converting to 2NF

In this second conversion step we are going to use unfolding to create a grammar in which the heads of the production rules consist only of auxiliary symbols Z_i. Unfolding is defined as follows:

Definition 7 (Unfolding) *The unfolding of a non-terminal N is accomplished by replacing every occurence of N in the right-hand sides of other production rules in the grammar by the right-hand side of N.*

We start by unfolding all applications of C:
$Z_2 \rightarrow c④③$

Continuing by unfolding all applications of A we get:
$Z_0 \rightarrow aZ_1⓪$
$B \rightarrow b②$
$B \rightarrow aZ_1Z_2$

Finally, unfolding all applications of B yields:
$Z_1 \rightarrow b②①$
$Z_1 \rightarrow aZ_1Z_2①$

We throw away the original and intermediate production rules for S, A, B and C, and also the intermediate Z_i production rules, keeping only the final production rules. We have now transformed the original grammar G to G', consisting only of auxiliary symbols Z_i:
$Z_0 \rightarrow aZ_1⓪$
$Z_1 \rightarrow b②①$
$Z_1 \rightarrow aZ_1Z_2①$
$Z_2 \rightarrow c④③$

Converting to 3NF

Note that *G'* is, by definition 6, already in 3NF, because no two alternatives of any production rule start with the same symbol. So we don't need this third conversion step. In case there are alternatives of a production rule A which do start with the same symbol, we can apply **full left factoring** (as defined in definition 8) **before** the 2NF construction. **After** all *non-terminals* have been unfolded (as described by definition 7), we apply full left factoring to all *terminals* in all productions. By applying this method, we get a grammar which conforms to 3NF (definition 6).

Definition 8 (Left factoring) *Let $\mathcal{G}$ be a CFG that contains one or more productions of the form as seen in the table below (where some of the w_i may be empty). Then left factoring extracts the common part v into a new production using a new nonterminal Z.*

$A \to vw_1$		$A \to vZ$
$A \to vw_2$		$Z \to w_1$
$A \to vw_3$	$\rightsquigarrow$	$Z \to w_2$
$\vdots$		$\vdots$
$A \to vw_n$		$Z \to w_n$

Full left factoring *is defined as left factoring with subsequent unfolding of the leading non-terminal (if such a non-terminal is generated).*

In the next subsection we are going to compare a recursive descent parser for *G'* to the standard LR(0)-parser (push-down automaton) for *G'*.

			4	3		8	7		
			b	B		c	C		
		6	6	6	5	5	5	3	
		a	a	a	A	A	A	B	
	2	2	2	2	2	2	2	2	1
	a	a	a	a	a	a	a	a	A
0	0	0	0	0	0	0	0	0	0

Figure 3: Sequence of stack contents when parsing `aabc` using the push-down automaton for *G*.

6.3 Comparison of the classical and the alternative parser

First consider the push-down automaton for *G*. It has a stack on which it pushes states and symbols while searching for a rule to reduce the symbols on the stack to. We will use the standard push-down automaton algorithm from [GBJL00]. When we parse the string `aabc` we get the sequence of stack contents which is shown in figure 3 (from left to right, and the top of the stack is at the top of the picture). The parse tree as constructed by this push-down automaton is shown step by step in figure 4. Consider now the alternative, recursive descent parser. This parser is implemented as a set of procedures which recursively call each other as defined by the production rules (see section 4.1).

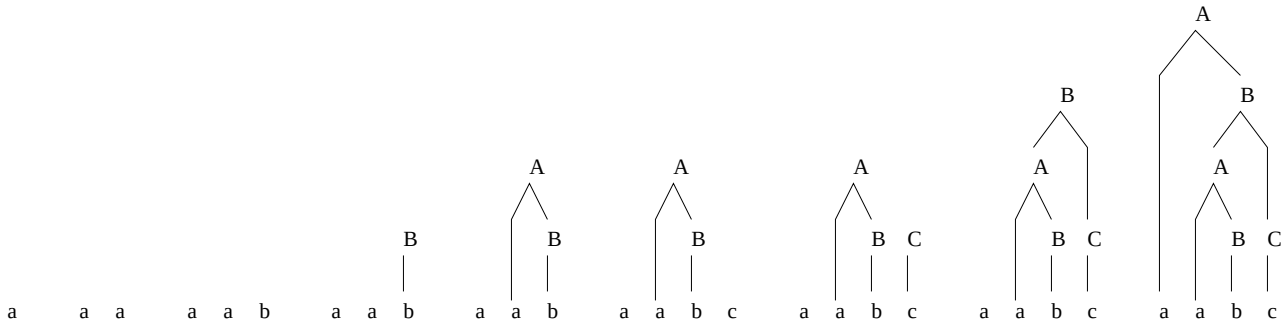


Figure 4: The parse tree constructed by the push-down automaton for G.

We can view the calls from the various procedures as a stack, and for parsing the string `aabc`, this stack initially looks like:
Z_0(`aabc`)

This means, procedure Z_0 is called with input `aabc`. When we continue tracing the procedure calls, we get the following sequence of stack contents:
aZ_1 ⓪ (`abc`)
aaZ_1Z_2 ① ⓪(`bc`)
aab ② ①Z_2 ① (`c`)
aab ② ①c ④ ③ ① ⓪

We can see the action symbols are still in the string resulted by this parser. Remember that the action symbols *derive* ϵ (the empty string), so actually the parsed string is `aabc`, which is, of course, correct. But the action symbols *carry more information*, they indicate which production rule from G was used to derive the preceeding terminal symbol(s). Using the action symbols, we can construct a parse tree. For example, when we have parsed the `b` from the input string, the procedure for the action symbol ② gets called. The symbol tells us "Reduce using $B \to b$", so we can reduce the `b` we've just parsed to B, and continue parsing. The next procedure which gets called is the one for ①, which says "Reduce using $A \to aB$". It turns out that the way the parse tree is constructed using this method, and the decisions the parser makes, are *exactly the same* as the push-down automaton for G, so the resulting parse tree looks exactly like the one in figure 4.

7 Theoretical Results

We have shown in our example that a traditional push-down parser for G, and a recursive descent parser for G', are equivalent in their computation processes. Formally, this is defined in [Pep99] as the "Main theorem":

Theorem 1 (Main theorem) *Consider a grammar $\mathcal{G}$ and its transformed 3NF version $\mathcal{G}'$. The original grammar is LR(k) if-and-only-if the transformed grammar $\mathcal{G}'$ is LL(k).*

Note that the reverse also holds: a non-LR(k) grammar will be non-LL(k) when transformed. In [Pep99] this theorem is proved, but we only list the most important aspects here:

- There is a strong relationship between LL and LR parsers.
- The resulting parser has the power of a LR parser, but (almost) the efficiency of a LALR parser.

A problem with the approach in [Pep99] is that the 3NF construction in its standard form does not always terminate. However solutions are described in [Pep99].

8 Conclusion

In the previous sections, we have shown that a LR(k) grammar can be parsed with LL(k) ease. The resulting parser is almost as efficient as a LALR(k) parser. The example we used in the section 6 is very simple, as its grammar is LR(0). But the parsing method actually works for all LR(k) and LL(k) context-free grammars. Special look-ahead non-terminals can be added (just like the action symbols ①), which derive ϵ as well. Error signalling/recovery can be added also.

We believe this method for LR parsing using LL techniques is easier than the classical approach, since each step in the parser creation is simple and yields a natural implementation from grammar to code. Also, applying the grammar transformation algorithms is less time consuming than first using the "Sets of Items"-method, deriving the ACTION/GOTO table and finally creating a table-driven parser for that ACTION/GOTO table.

But can we forget about the "Sets of Items"-method? We cannot, because *scanners* (also very important in compiler construction) also rely on a (modified) version of this technique. So the method remains important. Another problem with the alternative method is the fact that the grammar transformation procedures may not always terminate. Although solutions have been found, one might prefer to use the "Sets of Items" construction, which always does terminate.

References

[Bea82] John C. Beatty. On the Relationship Between the LL(1) and LR(1) Grammars. *Journal of the Association for Computing Machinery*, 29(4), 1982.

[GBJL00] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. John Wiley & Sons, 2000.

[Pep99] Peter Pepper. LR Parsing = Grammar Transformation + LL Parsing. *Technical Report*, (99-5), 1999.

[Pij93] Wim Pijls. Unifying LL and LR parsing. 1993.

[SB95] James P. Schmeiser and David T. Barnard. Producing a Top-Down Parse Order with Bottom-Up Parsing. *Technical Report*, (95-378), 1995.

[Tom] Mati Tombak. Formal languages and compilers.

[WM95] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison-Wesley, 1995.

An overview of several types of noise removal

Gijs Noorlander & Niels Maneschijn

jan 6^{th}, 2004

1 Abstract

After a short introduction into the characteristics of noise, an overview is given of several methods of noise reduction, with the purpose of improving upon edge detection. Also, the possibility of suppressing the effects of noise directly in the edge detection process is explored.

2 Introduction

The goal of this article is to explore different ways of removing noise from images, with the intended purpose of improving upon methods of edge detection.
Images and logos can be represented in many different ways, depending on the technique used to print the image. Normally we are used to see images captured with a camera. These images have a very large amount of (colour) shades. Some printing techniques however are not suitable for displaying shades.
When such a technique is to be used, you have to convert your image to an image containing only the available shades (this will often be a small number, like two or three shades). Printing techniques like these are often used to print on very large surfaces, like a flag. Therefore the image should have an almost infinite resolution, which can be represented by a vector format. A vector describes the outer lines of a surface, used in the image.
To make a graphical designer's work easier, one could try using the computer to detect the different edges and surfaces automatically and generate the vectors describing the image using an automated process.
Figure (1) is an example to show the difference between the human interpretation of an image and how the computer will trace the same image. The manually traced image will often be significantly smaller in the amount of objects used and therefore faster to process.

original	manually traced	automated trace	automated trace
320*200 pixels	20 vector objects	80 vector objects	135 vector objects

Figure 1: Different approaches for tracing an image

In reality the source picture may contain a lot of detail and noise which can make the resulting vector-image too complex to handle (e.g. the resulting vector image may contain as many as 100.000 objects, which have to be redrawn each time the image is edited).

One way of removing noise from images is using connected operators for detecting area closings and openings, for example using the Max-Tree method. In this article, we will also suggest other methods for removing noise. After an overview of these methods, we will discuss how edges are detected, and how the effects of noise can be minimized directly by choosing the right parameters for the edge detection.

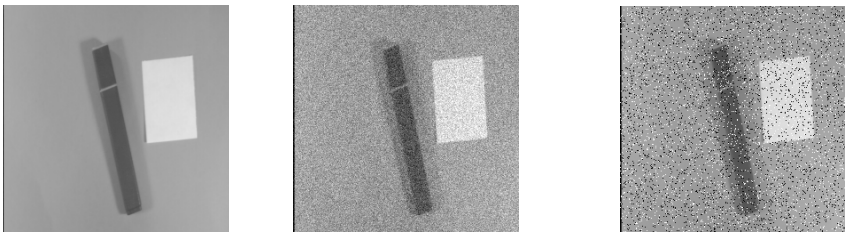
3 What is noise?

To understand the different methods of noise removal, one has to know some characteristics of noise.
A simple description of noise is "Random fluctuations in a stream of data". It can be characterized by both its amplitude and its spectral distribution. This spectral distribution can be constant in the whole frequency band (white noise), or it can be more apparent in certain frequency bands.

There are several properties of noise, which together characterisize a specific noise:

- **Additive noise** Random noise $n(i,j)$ added to pixel value $I(i,j)$ yields a new image
  $\hat{I}(i,j) = I(i,j) + n(i,j)$
  The noise $n(i,j)$ is often zero-mean and described by its variance σ_n^2. The impact of the noise on the image is often described by the *signal-to-noise ratio* (SNR), which is given by SNR= $\frac{\sigma_s}{\sigma_n} = \sqrt{\frac{\sigma_f^2}{\sigma_n^2} - 1}$ where σ_s^2 and σ_f^2 are the variances of the true image and the recorded image, respectively.
- **Gaussian noise** Each pixel in the noisy image is the sum of the true pixel value and a random, Gaussian distributed noise value. This noise value has a *zero-mean*. The Gaussian distribution is $p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$
- **Impulsive noise** - also called *peak*, *spot*, or *salt and pepper* noise, caused by transmission errors, faulty image sensor sites, etc.
  The corrupted pixels are either set to the maximum value (which looks like snow in the image) or have single bits flipped over. In some cases, single pixels are set alternatively to zero or to the maximum value, giving the image a 'salt and pepper'-like appearance. Unaffected pixels always remain unchanged. The noise is usually quantified by the percentage of pixels which are corrupted.

These types of noise are considered to be independent of the image-data.



Original Gaussian noise ($\sigma^2 = 20$) Impulsive noise ($\sigma^2 = 20$)

Figure 2: Examples of Gaussian- and Impulsive noise

4 Noise reduction techniques

To suppress this noise, several noise reduction techniques are available. We will discuss a few common ones in short.

4.1 Connected Operators using the Max-Tree method

Description: The idea of connected operators is to divide the objects in an image in zones with the same intensity value or colour, and sorting these zones according to their sizes. This way, the structure of objects shown in a picture can be reconstructed. By removing the smallest objects it is possible to remove noise. A convenient method for sorting is using a tree, which is used in the 'Max-Tree' method.

How it works: The datastructure used in this method is a tree. In this tree, groups of pixels are inserted, in such a way that the largest object, or zone of pixels of (approximately) the same colour, is at the root of the tree. The zones are defined as groups of pixels having intensities within a certain (usually small) threshold. The image can be reconstructed by reading back all zones, starting from the root of the tree, and adding these to the target image. This way, smaller objects are painted on top of the background. Because the smallest objects, including these consisting of noise, are located at the leaves of the tree it is easily possible to remove these objects by removing the leaves of the tree representing objects beneath a certain size threshold.This method of noise removal is mainly applicable for impulsive noise.[11][12][13]

4.2 Conservative Smoothing

Description:
Conservative smoothing is a noise reduction technique that derives its name from the fact that it employs a simple, fast filtering algorithm that sacrifices noise suppression power in order to preserve the high spatial frequency detail (e.g. sharp edges) in an image. It is explicitly designed to remove noise spikes — i.e. isolated pixels of exceptionally low or high pixel intensity (e.g. salt and pepper noise) and is, therefore, less effective at removing additive noise (e.g. Gaussian noise) from an image.[4][9][14]

How it works:
Like most noise filters, conservative smoothing operates on the assumption that noise has a high spatial frequency and, therefore, can be attenuated by a local

operation which makes each pixel's intensity roughly consistent with those of its nearest neighbours. However, whereas mean filtering accomplishes this by averaging local intensities and median filtering by a non-linear rank selection technique, conservative smoothing simply ensures that each pixel's intensity is bounded within the range of intensities defined by its neighbours.

4.3 Crimmins Speckle Removal

Description:
Crimmins Speckle Removal[5] reduces speckle from an image using the Crimmins complementary hulling algorithm. The algorithm has been specifically designed to reduce the intensity of salt and pepper noise in an image. Increased iterations of the algorithm yield increased levels of noise removal, but also introduce a significant amount of blurring of high frequency details.

How it works:
Crimmins Speckle Removal works by passing an image through a speckle removing filter which uses the complementary hulling technique (raising pixels that are darker than their surrounding neighbours, then complementarily lowering pixels that are brighter than their surrounding neighbours) to reduce the speckle index of that image. The algorithm uses a non-linear noise reduction technique which compares the intensity of each pixel in an image with those of its 8 nearest neighbours and, based upon the relative values, increments or decrements the value of the pixel in question such that it becomes more in line with its surroundings. The noisy pixel alteration (and detection) procedure used by Crimmins is more complicated than the ranking procedure used by the non-linear median filter. It involves a series of pairwise operations in which the value of the 'middle' pixel within each neighbourhood window is compared, in turn, with each set of neighbors (N-S, E-W, NW-SE, NE-SW) in a search for intensity spikes.

4.4 Gaussian smoothing

Description:
The Gaussian smoothing operator is a 2-D convolution [1] operator that is used to 'blur' images and remove detail and noise. In this sense it is similar to the mean filter, but it uses a different kernel that represents the shape of a Gaussian ('bell-shaped') hump. This kernel has some special properties which are detailed below.[6][7]

How it works:
The idea of Gaussian smoothing is to use this 2-D distribution as a 'point-spread' function, and this is achieved by convolution. Since the image is stored as a collection of discrete pixels we need to produce a discrete approximation to the Gaussian function before we can perform the convolution. In theory, the Gaussian distribution is non-zero everywhere, which would require an infinitely large convolution kernel, but in practice it is effectively zero at more than about three standard deviations from the mean, and so we can truncate the kernel at

[1] Convolution provides a way of 'multiplying together' two arrays of numbers, generally of different sizes, but of the same dimensionality, to produce a third array of numbers of the same dimensionality. This can be used in image processing to implement operators whose output pixel values are simple linear combinations of certain input pixel values.

this point. Figure (3) shows a suitable integer-valued convolution kernel that approximates a Gaussian with a sigma of 1.0.

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Figure 3: Discrete approximation to Gaussian function with $\sigma = 1.0$

4.5 Mean filtering

Description:

Mean filtering is a simple, intuitive and easy to implement method of smoothing images, i.e. reducing the amount of intensity variation between one pixel and its neighbours. It is often used to reduce noise in images.

How it works:

The idea of mean filtering is simply to replace each pixel value in an image with the mean ('average') value of its neighbors, including itself. This has the effect of eliminating pixel values which are unrepresentative of their surroundings. Mean filtering is usually thought of as a convolution filter. Like other convolutions it is based around a kernel, which represents the shape and size of the neighborhood to be sampled when calculating the mean. Often a 3x3 square kernel is used, as shown in Figure 4, although larger kernels (e.g. 5x5 squares) can be used for more severe smoothing. (Note that a small kernel can be applied more than once in order to produce a similar but not identical effect as a single pass with a large kernel).

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

Figure 4: 3x3 averaging kernel often used in *mean filtering*

Computing the straightforward convolution of an image with kernel in figure (4) carries out the mean filtering process.

Common variants:
Variations on the mean smoothing filter discussed here include Threshold Averaging in which smoothing is applied subject to the condition that the center pixel value is changed only if the difference between its original value and the average value is greater than a preset threshold. This has the effect that noise is smoothed with a less dramatic loss in image detail.[14] [6] [4]

5 Edge detectors

It is also possible to directly remove noise when detecting edges. This way an eventual first noise removal step can be skipped.
To understand how noise can be removed while performing edge detection, it is important to know how edge detection works. In this section we will discuss the principle of edge detection, followed by the implementation using a 'zero crossing detector'. At the end of the section we will discuss how the smoothing parameters of the zero crossing detector can be used to directly remove the effect of noise.

Edges are regions in the image with strong intensity contrast. Since edges often occur at image locations representing object boundaries, edge detection is extensively used in image segmentation when we want to divide the image into areas corresponding to different objects. Representing an image by its edges has the further advantage that the amount of data is reduced significantly while retaining most of the image information.
Since edges consist of mainly high frequencies, we can, in theory, detect edges by applying a highpass frequency filter in the Fourier domain or by convolving the image with an appropriate kernel in the spatial domain. In practice, edge detection is performed in the spatial domain, because it is computationally less expensive and often yields better results.
Since edges correspond to strong illumination gradients, we can highlight them by calculating the derivatives of the image. This is illustrated for the one-dimensional case in Figure 1.

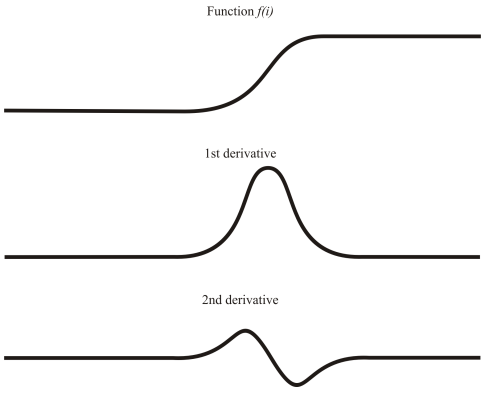


Figure 5: 1^{st} and 2^{nd} derivative of an edge illustrated in one dimension

We can see that the position of the edge can be estimated with the maximum

of the 1^{st} derivative or with the zero-crossing of the 2^{nd} derivative. Therefore we want to find a technique to calculate the derivative of a two-dimensional image. For a discrete one-dimensional function $f(i)$, the first derivative can be approximated by $\frac{d\ f(i)}{d(i)} = f(i+1) - f(i)$
Calculating this formula is equivalent to convolving the function with `[-1 1]`. Similarly the 2^{nd} derivative can be estimated by convolving $f(i)$ with `[1 -2 1]`. Different edge detection kernels which are based on the above formula enable us to calculate either the 1^{st} or the 2^{nd} derivative of a two-dimensional image. There are two common approaches to estimate the 1^{st} derivative in a two-dimensional image, *Prewitt compass edge detection* and *gradient edge detection*. Prewitt compass edge detection involves convolving the image with a set of (usually 8) kernels, each of which is sensitive to a different edge orientation. The kernel producing the maximum response at a pixel location determines the edge magnitude and orientation. Different sets of kernels might be used: examples include Prewitt, Sobel, Kirsch and Robinson kernels.
Gradient edge detection is the second and more widely used technique. Here, the image is convolved with only two kernels, one estimating the gradient in the x-direction, G_x, the other the gradient in the y-direction, G_y. The absolute gradient magnitude is then given by $|G| = \sqrt{G_x^2 + G_y^2}$ and is often approximated with $|G| = |G_x| + |G_y|$.
In many implementations, the gradient magnitude is the only output of a gradient edge detector, however the edge orientation might be calculated with $\theta = arctan(\frac{G_y}{G_x}) - \frac{3\pi}{4}$.
The most common kernels used for the gradient edge detector are the Sobel[2], Roberts Cross[1] and Prewitt[3] operators. After having calculated the magnitude of the 1^{st} derivative, we now have to identify those pixels corresponding to an edge. The easiest way is to threshold the gradient image, assuming that all pixels having a local gradient above the threshold must represent an edge. An alternative technique is to look for local maxima in the gradient image, thus producing one pixel wide edges. A more sophisticated technique is used by the Canny edge detector. It first applies a gradient edge detector to the image and then finds the edge pixels using non-maximal suppression and hysteresis tracking.
An operator based on the 2^{nd} derivative of an image is the Marr edge detector, also known as zero crossing detector. Here, the 2^{nd} derivative is calculated using a Laplacian of Gaussian (LoG) filter. The Laplacian has the advantage that it is an isotropic measure of the 2^{nd} derivative of an image, i.e. the edge magnitude is obtained independently from the edge orientation by convolving the image with only one kernel. The edge positions are then given by the zero-crossings in the LoG image. The scale of the edges which are to be detected can be controlled by changing the variance of the Gaussian.
A general problem for edge detection is its sensitivity to noise, the reason being that calculating the derivative in the spatial domain corresponds to accentuating high frequencies and hence magnifying noise. This problem is addressed in the Canny and Marr operators by convolving the image with a smoothing operator (Gaussian) before calculating the derivative.

5.1 Edge detection using a Zero Crossing Detector

(or *Marr* edge detector, *Laplacian of Gaussian (LoG)* edge detector)

Description:

The zero crossing detector looks for places in the Laplacian of an image where the value of the Laplacian passes through zero — i.e. points where the Laplacian changes sign. Such points often occur at 'edges' in images — i.e. points where the intensity of the image changes rapidly, but they also occur at places that are not as easy to associate with edges. It is best to think of the zero crossing detector as some sort of feature detector rather than as a specific edge detector. Zero crossings always lie on closed contours, and so the output from the zero crossing detector is usually a binary image with single pixel thickness lines showing the positions of the zero crossing points.

The starting point for the zero crossing detector is an image which has been filtered using the Laplacian of Gaussian filter. The zero crossings that result are strongly influenced by the size of the Gaussian used for the smoothing stage of this operator. As the smoothing is increased, fewer and fewer zero crossing contours will be found, and those that do remain will correspond to features of larger and larger scale in the image.[8]

How it works:

The core of the zero crossing detector is the Laplacian or Gaussian filter, some knowledge of that operator is assumed here. As described above, 'edges' in images give rise to zero crossings in the LoG output. For instance, Figure (6) shows the response of a 1-D LoG filter to a step edge in the image.

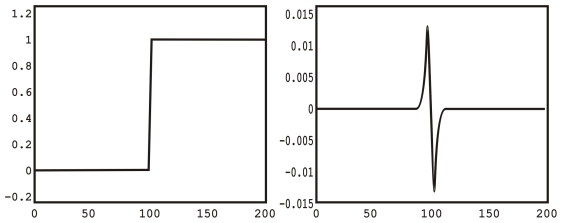


Figure 6: Response of 1-D LoG filter to a step edge. The left hand graph shows a 1-D image, 200 pixels wide, containing a step edge. The right hand graph shows the response of a 1-D LoG filter with a Gaussian standard deviation of 3 pixels.

The behavior of the LoG zero crossing edge detector is largely governed by the standard deviation of the Gaussian used in the LoG filter. The higher this value is set, the more small features will be smoothed out of existence, and hence fewer zero crossings will be produced. Hence, this parameter can be set to remove unwanted detail or noise as desired. The idea that at different smoothing levels different sized features become prominent is referred to as 'scale'.

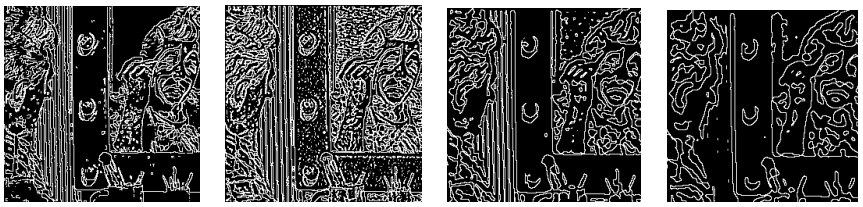
5.2 Effects of smoothing and usage for noise reduction

We illustrate this effect using figure 7(a), which contains detail at a number of different scales.

(a) original (b) LoG $\sigma^2 = 1.0$ (c) LoG $\sigma^2 = 2.0$ (d) LoG $\sigma^2 = 3.0$

Figure 7: LoG filter with Gaussian σ^2

Figure 7(b) is the result of applying a LoG filter with Gaussian standard deviation 1.0. Note that in this and in the following LoG output images, the true output contains negative pixel values. For display purposes the graylevels have been offset so that displayed graylevel 128 corresponds to an actual value of zero, and rescaled to make the image variation clearer. Since we are only interested in zero crossings this rescaling is unimportant.



(a) ZC $\sigma^2 = 1.0$ and threshold (b) ZC $\sigma^2 = 1.0$ (c) ZC $\sigma^2 = 2.0$ (d) ZC $\sigma^2 = 3.0$

Figure 8: Zero crossings with Gaussian σ^2

Figure 8(b) shows the zero crossings from this image. Note the large number of minor features detected, which are mostly due to noise or very faint detail. This smoothing corresponds to a fine 'scale'.

Figure 7(c) is the result of applying a LoG filter with Gaussian standard deviation 2.0.

Figure 8(c) shows the zero crossings. Note that there are far fewer detected crossings, and that those that remain are largely due to recognizable edges in

the image. The thin vertical stripes on the wall, for example, are clearly visible.

Figure 7(d) is the output from a LoG filter with Gaussian standard deviation 3.0. This corresponds to quite a coarse 'scale'.

Figure 8(d) is the zero crossings in this image. Note how only the strongest contours remain, due to the heavy smoothing. In particular, note how the thin vertical stripes on the wall no longer give rise to many zero crossings.

All edges detected by the zero crossing detector are in the form of closed curves in the same way that contour lines on a map are always closed. The only exception to this is where the curve goes off the edge of the image.

Since the LoG filter is calculating a second derivative of the image, it is quite susceptible to noise, particularly if the standard deviation of the smoothing Gaussian is small. Thus, it is common to see lots of spurious edges detected away from any obvious edges. One solution to this is to increase the smoothing of the Gaussian to preserve only strong edges. Another is to look at the gradient of the LoG at the zero crossing (i.e. the third derivative of the original image) and only keep zero crossings where this is above a certain threshold. This will tend to retain only the stronger edges, but it is sensitive to noise, since the third derivative will greatly amplify any high frequency noise in the image.

Figure 8(a) is similar to the image obtained with a standard deviation of 1.0, except that the zero crossing detector has been told to ignore zero crossings of shallow slope (in fact it ignores zero crossings where the pixel value difference across the crossing in the LoG output is less than 40). As a result, fewer spurious zero crossings have been detected. Note that, in this case, the zero crossings do not necessarily form closed contours.

Marr[10] has suggested that human visual systems use zero crossing detectors based on LoG filters at several different scales (Gaussian widths).

6 Conclusion

An overview is given of the characteristics of noise, and of several methods of noise reduction. It is also demonstrated that by choosing the right parameters, it is possible to reduce the effects of noise directly in an edge detection process.

The performance of edge detection algorithms can be substantially improved upon. By applying noise removal techniques, the amount of detected components can be reduced significantly. One major disadvantage of removing noise is the loss of detail.

One has to find a trade-off between the loss of detail and the accuracy of the edge detection. This ratio may be different for each situation, depending on the image-data and the desired level of detail.

Sometimes, the detection can be improved by applying noise removal, even though the image contains almost no noise. By doing so, it is possible to set the level of detail found by the edge detection.
It appears the best way is still to have the image as clean as possible in the original image, despite all the advanced noise removal techniques.

References

[1] http://www.ii.metu.edu.tr/~ ion528/demo/lectures/6/2/1/index.html.

[2] http://www.ii.metu.edu.tr/~ ion528/demo/lectures/6/2/2/index.html.

[3] http://www.ii.metu.edu.tr/~ ion528/demo/lectures/6/2/3/index.html.

[4] Boyle, R., and Thomas, R. *Computer Vision: A First Course*. Blackwell Scientific Publications, 32–34, 1988.

[5] Crimmins, T. The geometric filter for speckle reduction, may 1985.

[6] Davies, E. Machine vision: Theory, algorithms and practicalities, 1990.

[7] Gonzalez, R., and Woods, R. *Digital Image Processing*. Addison-Wesley Publishing Company, 191, 1992.

[8] Gonzalez, R., and Woods, R. *Digital Image Processing*. Addison-Wesley Publishing Company, 442, 1992.

[9] Jain, A. Fundamentals of digital image processing, 1986.

[10] Marr, D. Vision, 1982.

[11] Meijster, A., and Wilkinson, M. A comparison of algorithms for connected set openings and closings, april 2002.

[12] Monasse, P., and Guichard, F. Fast computation of a contrast-invariant image representation.

[13] P. Salembier, A. O., and Garrido, L. Anti-extensive connected operators for image and sequence processing, april 1998.

[14] Vernon, D. Machine vision, 1991.

A Survey of Bridging the Gap Between SE and HCI

Alex Westerhof, Martijn Kelder
University of Groningen, Department of Mathematics and Computing Science
a.c.westerhof@wing.rug.nl, m.kelder@wing.rug.nl

Abstract

The usability of many information systems is very poor. In this paper we describe the gap, which is the main cause of the lack of usability, between software engineering and human-computer interaction. First we outline the SE and the HCI disciplines. After that we describe the causes of the gap. Then we show three possible methods to bridge the gap. We discover that both SE and HCI are working to bridge this gap. We also see that in education they are trying to coordinate two courses on SE and HCI in order to realize a better cooperation.

1. Introduction

Information systems play an important role. A society without information systems is unthinkable. Therefore, as a matter of course, interaction between an information system and human users is very important.
Usability is a significant factor of the system's quality. The most cited attributes amongst authors in the usability field are [1]:

- learnability
- efficiency of use
- reliability
- satisfaction

Actually, the usability of many information systems is very poor [2]. Partly, this is the result of gaps between software engineers and human computer interaction engineers. Both disciplines design and implement information systems, from very different perspectives [7].
In this paper, we will look for a way to bridge the gap between SE and HCI. We try to answer the question:

How can the gap between software engineering and human-computer interaction be narrowed?

To answer this question, we first describe the two disciplines and the causes of the gap. Finally, we give a survey of three different methods to bridge the gap.

The goal of this paper is to give the reader a good impression of the current state of the research to bridge the gap between SE and HCI.

The rest of the paper is organized as follows. Section 2 outlines the software engineering discipline. Section 3 describes the human computer interaction discipline. In section 4, we describe the causes of the gap between the two disciplines. Section 5 gives solutions, which can help to bridge the gap. Section 6 concludes the paper.

2. Software Engineering

First, we will give a short overview of the software engineering discipline. We start with some definitions of software engineering [11]:

- The establishment and use of sound engineering principles (methods) in order to economically obtain software that is reliable and works on real machines.
- The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
- The technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.

We can identify some standard activities in a generic development process. We only mention activities affecting usability. This table is based on the SWEBOK (SoftWare Engineering Body Of Knowledge) [2].

Analysis (Requirements Engineering)	Req. Elicitation	
	Req. Analysis	Develop Product Concept
		Problem Understanding
		Modeling for Specifiaction of the Context of Use
	Req. Specification	
	Req. Validation	
Design	Interaction Design	
Evaluation	Usability Evaluation	Usability Testing
		Expert Evaluation
		Follow-Up Studies of Installed Systems

In order to develop useful and usable systems we have to understand user needs and contexts, represent them in user requirements and maintain a focus on user requirements throughout the development [3].

As said in the introduction, software is very important today. All software ultimately serves human needs and interests, hence provision for effective interaction between human users and software would reasonably be expected to be integral to all software engineering. Such has not been the case, historically. Until recently, software engineering methods and practice have focused primarily or exclusively on the internal structure and functioning of computer programs, leaving the interaction with users and the user interface that supports that interaction to other disciplines and professionals [4]. Usability is not addressed in software development as often as would be necessary to output highly usable software [5]. In some projects the user and customer requirements were not actually gathered or documented. Gathering user information was sometimes passive, methods were informal, and the gathered information was not documented at all [3].

When usability issues have to be solved, the reality is that average developers make the majority of the numerous decisions that determine the ultimate usability of a software product [2]. Software developers often identify usability with just the design of the graphical user interface, a part of the system that is developed at the end of the software development process. This kind of approach is responsible for the development of systems with a very low usability level [2]. Most usability issues are only discovered late in the development process, during testing and deployment. This late detection of usability issues is largely due to the fact that in order to do a usability evaluation, it is necessary to have both a working system and a representative set of users present. This evaluation can only be done at the end of the design process. It is therefore expensive to go back and make changes at this stage [1].

At the same time software engineering professionals want to produce quality products, which will be used by end users, but their syllabus is not driven by a focus on usability [6]. Mostly, the object oriented analysis and design (OOA/D) methodology is used. The methodology uses the UML notation and applies many of the same ideas as the Unified Process. The OOA/D methodology divides analysis and design into five main steps: definition of system scope and purpose, development of a model of the system's problem domain, analysis and definition of functionality and interface requirements, architectural design, and object design. The methodology emphasizes a layered architecture and the use of analysis and design patterns. It explicitly recognizes that some of the activities in the methodology – particularly definition of system scope and purpose, and determination of interaction requirements and functionality – depend on skills and techniques not covered by the methodology. These skills and techniques are briefly mentioned and some short examples of their use are given but in general they are treated as belonging to activities 'outside' the methodology [7].

3. Human Computer Interaction

HCI in the large is an interdisciplinary area that is concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them. HCI is emerging as a specialty concern within several disciplines, each with different emphases:

- Computer Science (application design and engineering of human interfaces)
- Psychology (the application of theories of cognitive processes and the empirical analysis of user behavior)

- Sociology and Anthropology (interactions between technology, work and organization)
- Industrial Design (interactive products)

HCI is concerned with:

- The joint performance of tasks by humans and machines
- The structure of communication between human and machine
- Human capabilities to use machines (including the learnability of interfaces)
- Algorithms and programming of the interface itself
- Engineering concerns that arise in designing and building interfaces
- The process of specification, design and implementation of interfaces
- Design trade-offs

Because HCI studies a human and a machine in communication, it draws from supporting knowledge on both the machine and the human side. On the machine side, techniques in computer graphics, operating systems, programming languages, and development environments are relevant. On the human side, communication theory, graphic and industrial design disciplines, linguistics, social sciences, cognitive psychology, and human performance are relevant. And, of course, engineering and design methods are relevant [8].
The engineering of the interfaces between humans and software requires solving design problems in three intimately interrelated areas: architecture, presentation, and interaction. The architecture of the interface refers to the overall and large-scale organization of the interface, that is, the way in which the interface as a whole is partitioned into distinct and recognizable regions or parts, how these parts are related and interconnected, and how the user navigates among these parts. Presentation design addresses the specification of how information is presented to users by visual, audible, or other means. Interaction design addresses the specification of the means by which users interact with a system and includes the organization of discrete steps in processes, the selection or specification of gestures or idioms for interaction, the sequencing of actions, and workflow. These three aspects of interface design interact strongly and are ultimately inseparable, even if some designers and methods strive to address them independently [4].

HCI engineers have to communicate with software engineers. For the HCI engineer it is important to understand the software engineer's attitude in order to be able to cope with the situation. This is less complicated for usability experts that have their roots in computer science than for those who come from graphics design or psychology. The latter may not speak the same language as the software engineers.
HCI engineers need competence and substantiality. They have to take into account that some colleagues have no conception of the techniques of usability engineering while at the same time maintaining their point of view. This is a challenge for HCI engineers. They have to continuously propagate the benefits of usability. They also have to explain the way in which they come to their design proposals in order to demonstrate a professional attitude [9].

4. Causes of the Gap

Software engineers and human computer interaction architects have to cooperate with each other to create a computer system. Usability is an important quality attribute. Today, we see computer systems that are less usable than they should be. This is partially the result of gaps between software engineers and human computer interaction engineers. In this section we will give the important causes for the gaps.

One of the causes of the gap is that it took some time before the two disciplines recognized the significance of each other and even adopted the approaches developed by each other. SE did not realize that usability is an important attribute of a software system. User’s needs were taken into serious consideration. Also HCI is changing, they look deeper into software development and software analysis issues than before [10].

Software engineering and human computer interaction both are modules, which are given to students who study computer science. The software engineering module doesn’t pay much attention to usability. Interaction between systems and human users are briefly mentioned, but in general they are treated as belonging to activities outside the course. The purpose of the human computer interaction module is to state that human interaction with systems and products require more than a flawless technical design [7].

It is difficult to transfer HCI techniques to the formalized SE processes. As said before, both fields speak different languages, and they approach software development from

a very distinct perspective. The HCI is multidisciplinary; HCI foundations come from the disciplines of psychology, sociology, industrial design, graphic design and etcetera. Software engineers have a typical engineering approach [2] [5]. The software process must be an iterative approach to use HCI techniques. It is almost impossible to create a correct design at once because the human side in human-computer interaction is very complex [7]. It is therefore not possible to use HCI techniques in non-iterative SE processes.

Traditionally, usability of a software system is evaluated at the end of the software engineering process. It is also a fact that quality attributes of a software system are, to a large extent determined by the architecture. The lack of an assessment to evaluate whether a given architecture meets the usability requirements is also bad for the usability of a software system [1] [4].

5. Bridging the Gap

A software development process should be iterative to be considered user-centered. It is the most important requirement for an existing development process to be a candidate for the introduction of usability techniques and activities. There are also two other characteristics: active user involvement and a proper understanding of user and task requirements [5]. We will describe three methods that intend to narrow the gap between SE and HCI. The backgrounds of the methods are respectively SE, HCI and education.

SE point of view

The usability of software has traditionally been evaluated on completed systems. Evaluating usability at completion introduces a great risk of wasting effort on software products that are not usable. A scenario based assessment approach has proven to be successful for assessing quality attributes such as modifiability and maintainability.
In [1] a scenario based assessment method is described to evaluate whether a given software architecture (provided usability) meets the usability requirements (required usability) based on the conjecture that scenario based assessment can also be applied for usability assessment. The advantage of this approach compared with traditional SE approaches is that you don't have to develop the complete system in order to check the usability.
The Scenariobased Architecture Level UsabiliTy Assessment (SALUTA) method consists of five main steps: goal selection, usage profile creation, software architecture description, scenario evaluation and interpretation. We will in short describe the five main steps.

- In the first step the results that will be delivered by the assessment are determined. The three goals that are distinguished are:
  - predict the level of usability
  - risk assessment
  - software architecture selection
- In the second step a usage profile describes usability requirements in terms of a set of usage scenarios. A usage scenario is defined as an interaction between the users and the system in a specific operation context.
- The third step, architecture description, concerns the information about the software architecture that is needed to perform the analysis. The result of this step is a description of the provided usability.
- Three different types of scenario evaluation techniques have been defined in the fourth step. The assessment techniques are complementary. Pattern based assessment, in which an expert assesses the architecture's support of usability, can be applied in most cases. Design decision based, where design decisions are evaluated, and use case map based assessment, in which use case maps are used to describe the architecture, may give additional information for the architectural support analysis.
- The last step is to interpret the results to draw conclusions concerning the software architecture. If the architecture proves to have sufficient support for all quality attributes the design process is ended. Otherwise architecture transformations or design decisions to improve certain quality attribute(s) need to be applied.

HCI point of view

Another model for narrowing the gap between SE and HCI is described in [4]. It is based on usage-centered design, which is a systematic, model driven approach to human interface engineering for software. It has evolved into a sophisticated process that has proved itself on projects of widely varying scope and scale in a variety of application areas.
Usage-centered design differs from its older and better-established counterpart user-centered design in several important ways. As the name suggests, the center of attention is not users but usage, that is, the tasks intended

by users and how these are accomplished. This difference in emphasis is reflected in differing practices that have a significant impact on the development life cycle and on integration with software engineering. Usage-centered design emerged directly from software engineering and particularly from object-oriented SE.

Precisely because usage-centered design is built around extensions and refinements to well established software engineering models and techniques, such as actors and use cases, integration with SE is more straightforward. Usage-centered design is conceived as an integrated concurrent engineering process aimed at producing an initial design that is essentially right in the first place. This is not to say that refinement and improvement through evaluation and feedback are not employed, but these are not the driving forces in the design process. Instead, usage-centered design is driven by interconnected models from which a final visual and interaction design are derived more or less directly by straightforward transformations. Iterative refinement applies more to the models from which the final design is derived than to the design.

The process is based on concurrent engineering, it is divided into concurrent but interdependent threads, one primarily focused on designing the human interface, and the other primarily focused on designing the internal software. A number of variants to this process and its models have been devised to suit various contexts and varying degrees of formality. In the usage-centered process, the presentation and interaction designs derive directly from the contents of the three tightly integrated core models:

- The *user role model* focuses exclusively on salient aspects of the relationships between users and the system as represented by the various roles users assume.
- In a *user task model* task cases are used. A task case is almost the same as a use case, but is a more abstract, modeling user intentions rather than actions and system responsibilities rather than responses.
- The *interface content model* consists of a set of abstract prototypes representing the contents of the various parts of the user interface and a navigation map representing the interconnections among all these parts.

The process models form the bridge between HCI and SE, providing traceability and interdependence and serving as the means for coordinating concurrent processes. For a usage-centered approach, non-human actors, referred to as system actors, are distinguished from user actors. User actors interact with the interface within the roles they play in relation to the system. Task cases support user roles. A given role typically requires a number of task cases.

On the SE side, the picture is more straightforward: system actors are supported by system use cases. Both task cases and system use cases become input for the object design, which must support them all to meet requirements. However, many details of the required software may not be determined until the presentation and interaction design are complete. The presentation and interaction designs are based on abstract prototypes of the various interaction contexts and the navigation map modeling the interrelationships among distinct contexts. Particular user interface components with specific behavior and appearance are chosen or designed to realize the abstract components. Abstract components are incorporated into abstract prototypes based on the particular tools and materials needed to realize each step in those task cases to be supported together within a given part of the user interface. The partitioning of the total user interface into subparts, as represented by the navigation map, is determined based on the interrelationships among task cases. A task case map in its most specific form models inclusions, extensions, and specializations relating task cases.

The domain model, which captures the core concepts of the application, is developed and refined concurrently and collaboratively throughout the process and serves to link the various design models. From this overview it should be clear that use cases, as task cases or system use cases, form the common thread that interconnects the various models and activities in the process. Indeed, task cases can directly guide the organization and contents of documentation and online help, thus providing uniform and comprehensive traceability throughout the delivered system.

The usage-centered software engineering process outlined here is not a proposal but an already well-established "industrial strength" process that has proved successful in numerous projects ranging up to 50+ person years completed by organizations around the world. Current areas of continued investigation include refinement in notation, compilation and elaboration of patterns based on canonical abstract components and task cases, and continued work on common theory and metrics, such as cohesion and coupling, underlying both object design and human interface design.

Perhaps most pressing is the need for tools that support usage-centered software engineering by incorporating its

models and exploiting their interconnections for flexible concurrent modeling and systematic requirements tracing.

Educational point of view

Yet another approach to narrowing the gap between SE and HCI is to teach students both interaction design and systems development in parallel. This method is described in [7]. There is no attempt made trying to share a new language between SE and HCI, instead creating a software prototype practices the coordination between the two approaches. The prototype makes the implicit assumptions behind the HCI and SE methods more explicit, e.g. assumptions of the power of modeling tools or the precision of usability design principles.

The integration of the two courses in one course curriculum faces several difficulties and challenges. Not only will it require a considerable effort to reconcile the differences in terminology and approaches to systems development, but there is also a risk that a joint or shared curriculum becomes disengaged from the underlying theories and perspectives of one or both of the disciplines. Practical software development on the other hand also requires a careful balancing of the two areas to avoid the danger of one approach becoming dominant. The challenge facing the teacher is of course how to combine the two disciplines in practical systems development. It is important to constantly emphasize the relations between the two areas when teaching. When teaching OOA/D, for example, emphasize that the use cases produced during interaction design and function specification are the visible end-result of a complex analysis of human users' work processes and interaction requirements. Conversely, when teaching the interaction design module emphasize how some of the artifacts produced when studying users and experimenting with various interface designs relate to the more 'formal' modeling activities in OOA/D.

It is the intention that the students are going to make two prototypes. A horizontal prototype, which demonstrates the user interface of a complete system. The prototype should be the end-result of systematically applying theories, tools and techniques taught in the interaction design module and it should conform to accepted user interface design principles. They also have to develop a vertical (running) prototype that implements a small subset of the (intended) final system's functionality. The prototype and its associated analysis and design documentation should be developed according to the methodology and conform to general software engineering analysis, design and programming principles. Thus, the vertical prototype demonstrates the students' ability to systematically apply the OOA/D methodology and produce a running program that is consistent with their analysis and design. The prototypes allow the students to apply the theories and techniques from each of the modules taught within the framework of an assignment of a reasonable size. By deliberately letting the students work in a 'grey zone' between the two clearly different approaches, we enable them -through their own practical experience- to realize how the fields of interaction design and software engineering together contribute to the construction of a system. Thus, they can either choose to make two independent prototypes and describe in words how they are related, or they can choose to develop one prototype that combines an implementation of a subset of the system's functionality with 'dummy' design of the user's interaction with a full system. Either way, they need to reflect on how to coordinate the two approaches.

6. Conclusions

In this paper we've tried to answer the question:

How can the gap between software engineering and human-computer interaction be narrowed?

We also gave the reader an impression of the current state of the research that has been done to bridge the gap between SE and HCI.

To achieve this, we outlined the SE and the HCI disciplines. After that we described the causes of the gap. Finally, we gave a survey of three different methods to bridge the gap.

We discovered that both SE and HCI are working together to design and implement computer systems with high usability. Despite the fact that both SE and HCI are working hard to bridge the gap, no ultimate solution has been found yet. So, a lot of research still has to be done to further narrow the gap.

7. References

[1] Eelke Folmer, Jilles van Gurp, Jan Bosch, "Scenario-based Assessment of Software Architecture Usability", University of Groningen, Department of Mathematics and Computing Science
http://www.se-hci.org/bridging/icse/accepted/09_Folmer_Groningen.pdf

[2] Xavier Ferre, Ana M. Moreno, "Improving Software Engineering Practice with HCI Aspects", Universidad Politecnica de Madrid
http://www.ls.fi.upm.es/udis/miembros/xavier/papers/integrHCI.pdf

[3] Sari Kujala, Marjo Kauppinen, Sanna Rekola, "Bridging the Gap between User Needs and User Requirements"
http://www.soberit.hut.fi/~skujala/PublicationIV.pdf

[4] Larry Constantine, Robert Biddle, James Noble, "Usage-Centered Design and Software Engineering: Models for Integration"
http://www.foruse.com/articles/models.pdf

[5] Xavier Ferre, "Integration of Usability Techniques into the Software Development Process", Universidad Politecnica de Madrid
http://www.ls.fi.upm.es/udis/miembros/xavier/papers/integration.pdf

[6] Ronan Fitzpatrick, "The Software Quality Star: A conceptual Model for the Software Quality Curriculum", Dublin Institute of Technology, School of Computing
http://www.se-hci.org/bridging/interact/Fitzpatrick.pdf

[7] Torkil Clemmensen, Jacob Nørbjerg, "Separation in Theory - Coordination in Practice", Copenhagen Business School, Department of Informatics
http://www.cbs.dk/staff/noerbjerg/publications_files/Clemmensen_Norbjerg_2003.pdf

[8] Hewett, Baecker, Card, Carey, Gasen, Mantei, Perlman, Strong, Verplank, "Curricula for Human-Computer Interaction", Association for Computing Machinery - Special Interest Group on Computer Human Interaction (ACM SIGCHI)
http://sigchi.org/cdg/cdg2.html

[9] Matthias Müller-Prove, "Mind the Gap! Software Engineers and Interaction Architects", Sun Microsystems
http://www.se-hci.org/bridging/interact/p100-101.pdf

[10] Effie Lai-Chong Law, "Bridging the HCI-SE Gap: Historical and Epistemological Perspectives", Eidgenössische Technische Hochschule Zürich, Computer Engineering and Networks Laboratory
http://www.se-hci.org/bridging/interact/Law.pdf

[11] Carnegie Mellon Software Engineering Institute, "What Is Software Engineering?"
http://www.sei.cmu.edu/about/overview/whatis.html

Introduction to various color segmentation methods and their applications

Wicher Visser Egbert van der Es

Department of Computing Science
State University of Groningen
Blauwborgje 3
Groningen 9747 AC
The Netherlands

{w.t.visser,e.van.der.es}@wing.rug.nl

Abstract
Segmentation is a technique that is widely used in low-level image processing. Images obtained from a camera need some processing before higher-level decision operations can be made. Often, objects have to be recognized which influence the behavior of the overall system.Examples of such operations are car steering systems, autonomous robots and security systems based on human visual features.
As a lot of the applications in which segmentation is involved are implemented in hardware, the segmentation methods used have to be robust; i.e. they have to be insensible to noise fluctuations. Due to this requirement, developers of such systems have to take into account the quality of the hardware and the performance of the selected methods and their properties.

Keywords: Color segmentation; Texture segmentation; Color space; Features; Object context; Histogram; Threshold; Split and merge; Region growing; Markov Random Field; Edge detection; Snakes; Neural network; Motion tracking; Real-time

1 Introduction

Segmentation is the process of partitioning an image into disjoint and homogeneous regions. These regions are generally called objects as they often represent real-life things or creatures [Lucchese and Mitra, 2001].

There has been done an enormous amount of research on image segmentation. However, this research limited itself to grey-scale images. Due to the immense (and fast) speed increase of computers and the price-drop of color imaging devices, color segmentation have only recently become interesting and feasible to implement. However, color images contain more information than grey-scale images and are therefore useful in areas where recognition or segmentation is a difficult issue. Examples of such situations are general robot vision [Heisele et al., 1997] and melanoma recognition [Hamarneh et al., 2000]. Nowadays, as computing power has grown beyond the required limits and color imaging devices have become cheaper, color segmentation has become an important field in computer vision. Fortunately, a lot of grey-value techniques can, with some adjustments, be applied to color images with great success.

This paper presents various color segmentation methods and their field of applications. An in-depth description of the algorithms will not be offered. Rather, we aim to present a general overview of the several techniques. The majority of techniques originate from the closely related area of

grey-scale segmentation. Most of these techniques can be easily adapted to cope with color information. However, this is outside the scope of this paper. In the first section, various applications of color segmentation methods are given. The second section discusses some preliminaries on color images. Section three contains various color segmentation methods grouped by basis of operation, such as edges or regions. We then discuss the use of some color segmentation techniques in motion tracking in section four. Hardware aspects of the various segmentation methods are treated in the fifth section.

Where color segmentation focuses on reducing the image to uniform colored regions, texture segmentation also tries to identify color patterns in the image.

2 Applications

In this section we will discuss the various applications of segmentation. Segmentation plays an important roll in low-level image processing. In virtually all situations where a computer needs to interpret an image, it is one of the most basic operations to be performed. The most important reason to apply color segmentation is to simplify the image at hand so that operations further on in the processing pipeline can focus on major objects in the scene. By removing as much insignificant features as possible a computer can initially interpret the image in a broad sense. When all the larger objects are located and maybe even identified in the abstract image, the computer can concentrate on the smaller objects in the non segmented image.

One of the areas where segmentation is important is robotics. Especially in cases where some autonomous behavior is expected and the robot needs to interpret its environment in order to orient itself and decide what his next move will be. A robot might want to know its position relative to other objects or find a specific object that needs to be handled. An interesting case where both apply is robocup, where multiple autonomous robots play a game of soccer against each other in two teams. The ball needs to be found within the playing field so that it can be approached and the direction in which it's rolling can be changed and the ball ultimately finds its way into the goal. The robot also needs to know its position in the playing field to find the goal and the position of his team mates and his opponents. Otherwise he wouldn't be able to prevent the ball from being lost to a member of the opposite team or pass the ball to one of his team mates. Of course you don't want the robots to run into each other or the boundaries of the field all the time. By marking the ball and the players in the field by specific colors and placing colored markers around the field, color segmentation can be used to achieve all this. But there are many more practical situations where robots need to have an idea on what is around them, for example scouting robots in the military.

In the medical world color segmentation can be helpful to process images captured by x-ray, MRI or ECG. The computer can aid humans by locating anomalies (like tumors), arteries, bones, bone fractures and organs and thereby making it easier for doctors to state a diagnosis and reducing the risk anomalies are overlooked. Another example where computer vision can reduce human effort and increase safety is in car steering systems. By keeping track of markings on the road a computer can correct the trajectory of the car to keep it on the road. By keeping track of the cars surrounding the car the computer can prevent collisions. One system that can already be found in use is a warning system which alerts the driver when the car is going off track. Color segmentation plays an important role here.

More applications of segmentation are in the security business. Especially interesting is the concept of face recognition. In the first stage computers try to locate faces in the picture. You can however go one step further by trying to recognize facial features. Using these features for a query in a face database might result in the identification of a certain person. This can be useful on the streets or on airports to recognize criminals or in a system where face recognition is used to provide limited access to secure areas.

Color segmentation is used in quality control of for instance fruit and vegetables. Because nature does not always provide us with the same high quality products a quality check is expected. By locating fruit on the transport belt and inspecting the color, rotten or immature ones, which are represented by a specific color region distinct from healthy fruit, can be picked out.

3 Preliminaries

3.1 Color space

A decision one has to make when using color segmentation is what color space to use. Making the right choice is important because it plays an important role in the quality of the final result of the segmentation. First has to be determined what color characteristics are the most important distinguishing factor between the color regions. Color regions that differ on these characteristics can easily be separated in the right color space, whereas in an inappropriate color space they might be lying very close together or even overlap. This will make it hard or even impossible to separate the color regions from each other, which results in poor segmentation.

The most common color spaces will be discussed briefly in this section. It would be out of the scope of this document to discuss all the different kinds of color spaces in use, simply because there are too many of them. The most common is the RGB space where the colors are represented by the red, green and blue components. This color space is related to the human visual system where there are three different kinds of photo-receptors on the retina whose responds are tuned to the wavelengths of these three colors. The same three color components are used to visualize images on displays and in digital cameras.

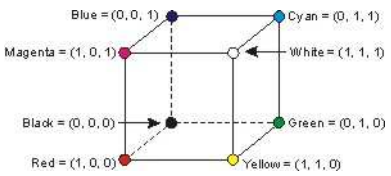


Figure 1: Cartesian coordinate system for RGB color space

There are drawbacks to color representations that are essentially the decomposition of the color in separate color components. This representation does not lend itself to distinguish between characteristics other than the intensity of the color components, unlike the perception of color of the human visual system. In this sense, the color can better be represented by hue, saturation and intensity. The HSI space is an well known example of this kind of representation. Using certain formulas RGB color space can be mapped upon HSI space and visa versa.

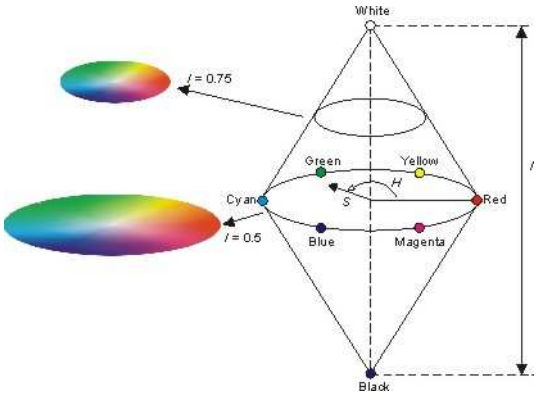


Figure 2: Cylindrical coordinate system for HSI color space

The HSV color space is also very common and strongly related to the HSI space since it uses only a slightly different color mapping.

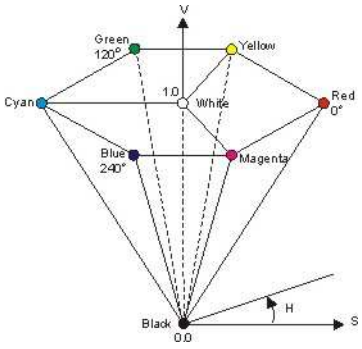


Figure 3: Cylindrical coordinate system for HSV color space

However, RGB and HSI are not perceptually uniform, meaning that differences perceived by the human eye do not result in similar distances in these color spaces. To solve this problem the uniform color spaces were introduced. The most common

uniform spaces are the Luv and Lab color spaces. There are several more color spaces that are two dimensional rather than three dimensional.

3.2 Features

Before segmenting the image into objects, it is needed to extract the various features on which the segmentation is based. Most often, color intensity or value is used as main feature. However, other features can be important as well. The location of a pixel relative to an other (possibly already defined object) can be a significant feature in case of subsegmentation (i.e. segmenting an object retrieved from previous segmentation, e.g. identifying nose, eyes and mouth in a face). Other features are the number of holes (character recognition), size of a uniform color region [Ganster et al., 2001] and the number of connected components.

The n features can be collected in a n-dimensional feature space. Significant features in the image correspond to high density regions in this space. To retrieve the objects from the image, the high density regions in the feature space should be found.

For large n, the feature space should be reduced to make it computationally efficient. However, reducing the feature space is only an option if there exist features that are linear dependent on the others. As the feature space is discrete rather than continuous, groups of feature vector can be created based on their similarity. Hereby, the range of the feature space is reduced [Comaniciu and Meer, 1997]. For instance, assume a 3-dimensional feature space consisting of the length, width and square of objects from which we know they are rectangular. In such a specific case, the square surface does not add any additional information, as it can be computed (it is linearly dependent) from the length and the width. It therefore can re removed without losing a lot of information.

3.3 Object context

The methods used for image segmentation depend largely on the context in which the application should function. If the application is used in situations where a static background is used, this prior knowledge can be used to create simpler and better applications. However, if the method has to function in a dynamic surrounding (e.g. an automatic car driving system [Heisele et al., 1997]) in which the background as well as the illumination varies, a more robust application has to be built.

Face recognition system often focus on skin color to segment the face from the background. It is easy if the developers can use a single color (band) such as pink to represent the skin color [Fritsch et al., 2002]. The application then only functions on white people, whereas negroids and Asians are treated as background. If their skin colors should be treated as well, the segmentation method has to be adjusted. The implementation therefore depends on the context of the application.

4 Segmentation methods

Various segmentation methods have been developed to segment objects in images. These methods can be classified into (more or less distinct) groups. Note that, besides the methods and groups presented in this paper, there are many more techniques that can be used to obtain the required result. We aim at summarizing the most frequent used methods, which form the basics of computer vision.

4.1 Global-based segmentation

In order to segment images properly, mostly some preprocessing is needed. For this case, histograms and thresholds are often used.

4.1.1 Histograms

A histogram is a graphic representation of the frequency distribution of a property of the image. For grey-scale images, a histogram of the grey value distribution can easily obtained by counting the number of pixels of a certain intensity (grey value). A histogram of a colored image can be produced in various ways, depending on the preferred feature. Assumed the pixels color are represented by HSV values, a cellular decomposition method (fig. 4) can be used to dynamically select the colors needed to process the image [Chen and Chen, 2001]. A hexagon represents the colors needed for processing. The two dimensions of the hexagon are represented by the hue and the saturation. The V (color value) component is divided into three

ranges, yielding 21 initial colors. For a pixel, the Euclidean distance between each pixel and the colors present in the hexagon is computed. The pixel is assigned the color which has the minimum distance to the pixel. The square error for each color is computed and the color with the largest error is decomposed into a new hexagon. Decomposition stops when the maximum level or the maximum number of colors is reached. After this splitting, similar colors are merged.

The dynamically obtained colors can then be used to create a histogram representing the color distribution in the image. If the histogram is normalized, statistical methods can be used to reason about the probability of an image fitting into a category.

Varying lightning conditions in images, such as shadows or reflections of light sources can be compensated for using histograms. Basically, such a method creates a uniform distribution of the intensity levels of the pixels in the image based on the intensity histogram [Fritsch et al., 2002]. Histograms can also be used as features for higher order segmentation methods.

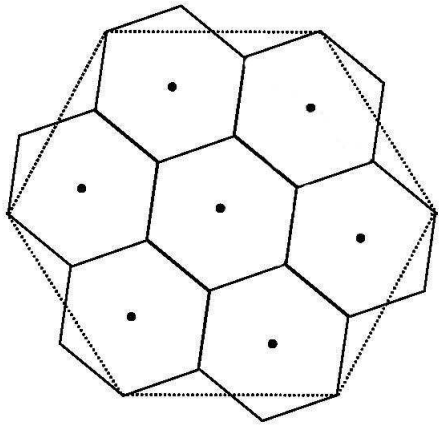


Figure 4: A color decomposed into a hexagonal structure with seven colors.

4.1.2 Thresholding

An easy and fast method to find objects in an image which have a single color (range) is global thresholding [Fritsch et al., 2002]. Color thresholding is mainly done by selecting bands in the three dimensional color space [Bruce et al., 2000]. Only those pixels that are within the selected range are filtered out; all the others are blackened. The resulting pixels may then represent objects. Unfortunately, this method is not very useful in daily situations where objects may consists of many separate colors. However, in laboratory setting and in special cases (such as a manufacturing process) this method can be effective.

In case of varying illumination in the image, standard thresholding is not an option. Here, dynamic thresholding can be used [Ganster et al., 2001]. The threshold level then depends on local properties of the image, such as brightness of a region. If the variation of illumination is strong, local thresholding will be a better strategy.

Histograms can be effectively used to select the color bands of the threshold. The regions of the color space which potentially represent an object can be identified by selecting a color band which histogram value is larger than the minimum expected size of the object.

4.2 Region-based segmentation

In region based algorithms new regions are formed by combining homogeneous pixels or regions. This is in contrast to edge based algorithms where regions are formed inside boundaries defined by differences between pixels. Region based algorithms fit mostly in one of the two categories being region growing techniques and split and merge techniques. Another popular method is Markov Random Fields, which is discussed as final topic.

4.2.1 Split and merge

The basic idea behind the split and merge technique is pretty easy to grasp. You start initially with an inhomogeneous area and split this area recursively until you are left with homogeneous partitions. After splitting the area, the neighboring homogeneous partitions are merged satisfying the requirement the two merged regions are still homogeneous. This process is repeated until no regions

can be merged together. The homogeneous condition can be based on the color information of the image. E.g., if the image is represented in the HSI color space, the intensity value I can be used to separate the bright and dark areas. For the splitting phase the quad-tree [de Berg et al., 2000] is a much used data-structure and for the merging phase the region adjacency graph [Tremeau and Colantoni, 2000] is very common. Nevertheless there are many other methods and structures in use. Some methods incorporate Delauney triangulations or Voronoi Diagrams [de Berg et al., 2000]. Other adjustments have been made to optimize the merging phase by using stochastic methods, to optimize smoothness and continuity. By searching for patterns during the splitting phase you can expand the method with texture segmentation capabilities.

4.2.2 Region growing

Using region growing, a region can be found by starting of with a region of one single point and continuously adding homogeneous neighboring points. This process goes on until no more pixels satisfy the homogeneous condition. Region growing is mostly used for single regions, but by combining subsequent growth processes one can grow regions until all points fall in a region. Region growing is a sequential process, thus the results depend on the order in which the points are processed. An advantages is that clusters are compact and, for sure, connected. For color images some advancements have been made to the algorithm. Examples are seed finding based on the color image gradient and region growing based on color and intensity information. Some algorithms take small gradual changes into account during the growing process, making the algorithm an edge and region based hybrid. Another way of growing regions is by using the watershed method [Angulo and Serra, 2003].

4.2.3 Markov Random Fields

In real-life, objects in an image are often represented by their texture. A Markov Random Field can be used to model the texture. Markov Random Fields can be used to characterize the statistical relationship between a pixel and its neighbors. A pixel is said to be part of a texture region if the features of its local neighborhood (random field) have a high enough probability to represent the targeted texture [Liévin and Luthon, 2001].

4.3 Edge-based segmentation

Another way of segmentation is the detection of edges surrounding regions. There are several edge detection methods called chromatic edge detectors or detectors that require gradient information by using the Sobel operator. There are also predictive models that try to predict change in color. Regions can contain colors that differ greatly, because if the change between the colors is gradually there may never be an edge detected between them. A disadvantage of this method is that some edges may be disconnected and that after the edge detection-phase you are not left with clean cut edges and have to add another linking phase in order to connect all dangling edges.

4.3.1 Snakes

A snake, also known as active contour model, is an energy minimizing curve which changes its shape under the control of internal and external forces. Snakes try to model objects in the image using edges representing the object. These edges are often based on the gradient of the intensity levels in the image. The minimum energy state of the snake should be the location of the edge to be detected. A snake consists of a string of connected vectors that can move through an image. The vectors however are unable to penetrate certain edges and are forced to hold their position. By using a shrinking or expanding ring of vectors you can detect region outlines. For snakes to work, the boundary of the targeted object should be inherently connected and smooth [Hamarneh et al., 2000] to a certain degree. Several optimizations exist where the number of vectors on the snake dynamically adapts to the situation at hand, or big snakes can split up into smaller snakes to detect multiple regions. Often, the gradient in each of the main color components (e.g. HSV) is computed and merged together to form a single 'color gradient' on which the snake operates.

If illumination is uniform throughout the image, object boundaries are well matched. However, if the illumination varies due to e.g. shadows or highlight, the color gradient does not represent the tar-

geted contour properly. One way to prevent this performance degradation is to apply snakes to the gradient of each color component and then merge the results into a unified snake [Schaub and Smith, 2003]. A different solution is to assign positive pressure values to regions that are statistically similar to a seed region.

Another problem arises when objects in an image do not have a color transition where an intensity gradient is found. When using intensity levels as the feature used by the snake, the image segmentation can be negatively affected by the imaging process. (e.g. shadows, shading or highlights). To overcome this problem, color invariant gradient information can be used for the contour modeling [Geversa et al., 1998].

4.4 Neural networks

Some segmentation methods are based on neural networks [Chen and Wang, 2001] [Ong et al., 2002b]. Neural networks consist of a large number of massively interconnected elementary processors or cells. They try to imitate the biological neural cells. Neural networks offer some important properties in pattern recognition: neural networks lend themselves to be parallelized, which makes very fast computational performances possible and is therefor suitable for real-time applications. Neural networks are also very reliable, because the performance of the network suffers only mildly from disturbances in a relatively small number of cells. Neural networks can also be programmed to adapt to changing situations.

A network has to be prepared for recognizing segments by training. In a supervised learning stage one has to have some a priori knowledge in order to give the network examples it can learn from. There are also unsupervised methods where the network learns by itself. Different methods exist, but most of the time pixels of the image are fed into the network which produces pixels with a limited color depth. Pixels with the same color can then be grouped into regions. The number of segments the network can produce is limited by the number of colors it can reproduce as output. Therefore it must be known at forehand what the maximum number of segments is that are to be found in the images before the network is designed.

5 Motion tracking

Snakes are well suited for motion tracking. Given the first image of the sequence, the snake of the targeted object is computed. The resulting snake is then put on the next image and the minimum energy state is again computed. One important constraint is that subsequent images do not differ to much, as snakes might then loose the object [Sobottka and Pitas, 1996].

Natural objects are often non-rigid: they do not have smooth or connected contours. Therefore, snakes cannot be properly used for this application. Another solution is to segment images based on regions. The initial color image is segmented into clusters of relatively the same color, also called fuzzy color clusters. For every consecutive image, a prototype, representing the center of the cluster, is shifted in the feature space (e.g. x and y coordinate of the prototype in the image) by clustering it [Heisele et al., 1997].

As we described in section 3.1, snakes can be color invariant. However, snakes are computationally expensive and can therefore mostly not be used in real-time applications. If the objects in the image suffer from intensity fluctuations independent of the color gradient, fuzzy color clusters are no option. Texture, if present, can then be used in addition to increase the segmentation performance. Markov Random Fields can be used for this method. Their joint probability density functions (PDF) are computed and based here-on the segmentation is performed [Liévin and Luthon, 2001].

6 Hardware aspects

Segmentation methods are often embedded in hardware or run in software on special purpose hardware. Therefore, to keep the cost small, these methods should be computationally cheap. To satisfy this requirement while keeping a robust and reliable method, extra attention to the design process has to be made. In this chapter we will first direct our attention towards real-time segmentation methods. Second, image capturing devices will be discussed.

6.1 Real-time

Real-time processing of data is important in fields where a constant flow of data exists which should be monitored continuously. Examples of such fields are robotics [Bruce et al., 2000], automatic car steering [Heisele et al., 1997] and object tracking [Batavia and Singh, 2001].

To assure real-time segmentation methods to be robust, multiple cues (i.e. multiple method) can be used [Ozyildiz et al., 2001]. With this technique, multiple segmentation methods are used which have the same purpose: recognizing the object. After segmentation, the various outcomes of the methods are merged into a single result. The goal of this approach is to compensate for the areas in which certain cues perform badly. The final result can therefore be represented as the maximum of the performances of every segmentation method given a certain input pattern.

Another useful technique is parallelization. Neural networks are a good example of this approach [Ong et al., 2002a]. As cells in each level of the network are independent of each other, the cells in each level can be arranged in parallel, optimizing it for speed. Similar, color segmentation can be made parallel. Assume a robot soccer game in which team mates, opponents, ball and goal are all represented by a uniform color. Although the objects in the image taken by a robot can be affected by lighting, they can be identified by looking at a color region. A threshold indicates whether a object belongs to a certain class. Inspecting the color of a pixel, it can be assigned to multiple groups by checking the thresholds for all classes in parallel. A blue pixel can therefore belong to a corner flag and a goal. Other features can then be used to select the correct class.

As hardware performance increases every year, segmentation methods currently out of scope of real-time processing (e.g. 3D object recognition) will be used for this purpose in the nearby future, making better and more complicated segmentation possible.

6.2 Static time processing

A lot of segmentation methods do not aim at real-time processing. Examples of fields in which real-time segmentation is not a requirement are face analysis systems [Sobottka and Pitas, 1996], lesion or melanoma detection [Hamarneh et al., 2000] and automatic photo enhancement. Applications using this type of processing do not need to be fast. However, consumers still demand their applications to be fast. Therefore, the trade-off between performance and speed is harder to make in contrast to real-time applications, where the limit is more apparent.

6.3 Capturing

Segmentation is applied on digital images using computers. There are different ways to capture these digital images from the real world. The choice which capture device to use depends on aspects as cost, required information, available technique and speed requirements. It is common to try to keep cost as low as possible and choose for the cheapest device. A relatively cheap device may however not produce images of the required quality which results in unsatisfactory computer vision capabilities. For example, in order to recognize text on number plates, the resolution of the image needs to be fine enough to capture the individual characters. A high resolution camera on a real-time processor might on the other hand overload the system. A limitation on the available number of devices is the required information. Trying to take images of a persons intestents with a normal digital camera is pointless, unless the person is pretty messed up. In this case you need a device that can penetrate the skin like x-rays or a MRI scanner. The available techniques determines how much choice you have.

The used capture device has big influence on the quality and sort of information you acquire and has therefore great influence on the decision what segmentation method to use. Normal digital photo cameras and x-ray cameras deliver totally different kind of images. In observation satellites one might have different separate cameras present for different wavelengths. Depending on noise, contrast, color space and the color distribution within this color space, some segmentation methods are preferable over other.

7 Conclusion

With this paper an attempt has been made to give an introduction to color and texture segmentation.

Besides a basic explanation of these concepts and some strongly related issues, most of the paper is dedicated to segmentation methods. It is not easy to give a broad overview and place these methods in clearly distinguishable groups. Many segmentation publications are focusing on very specific segmentation tasks where different techniques are combined to achieve optimal results in bounded cases. There is also still a lot of active research on this subject. Most of these specific or new methods are however variations or refinements on methods we have mentioned here or can be decomposed to these methods. Therefore we believe that this paper may contribute to the understanding of the basics of color and texture segmentation and aid the learning of more advanced and recent methods.

References

Angulo, J. and Serra, J. (2003). Color segmentation by ordered mergings. *IEEE conference*.

Batavia, P. and Singh, S. (2001). Obstacle detection using adaptive color segmentation and color stereo homography. *Proceedings of the IEEE international conference on robotics and automation*, 1–6.

Bruce, J., Balsch, T., and Veloso, M. (2000). Fast and inexpensive color image segmentation for interactive robots. 1–6.

Chen, K. and Chen, S. (2001). Color texture segmentation using feature distributions. *Pattern recognition letters*, 756–757 and 769–770.

Chen, K. and Wang, D. (2001). Image segmentation based on a dynamically coupled neural oscillator network.

Comaniciu, D. and Meer, P. (1997). Robust analysis of feature space: color image segmentation. 1–8.

de Berg, M., Schwarzkopf, O., van Kreveld, M., and Overmars, M. (2000). *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edn.

Fritsch, J., Lang, S., Kleinehagenbrock, M., Fink, G. A., and Sagerer, G. (2002). Improving adaptive skin color segmentation by incorporating results from face detection. 1–7.

Ganster, H., Pinz, A., Rohrer, R., Widling, E., Binder, M., and Kittler, H. (2001). Automated melanoma recognition. *IEEE transactions on medical imaging, vol. 20*, 233–239.

Geversa, T., Ghebreab, S., and Smeulders, A. (1998). Color invariant snakes. 1–11.

Hamarneh, G., chodorowski, A., and Gustavsson, T. (2000). Active contour models: applications to oral lesion detection in color images. *IEEE conference on systems, man and cybernetics*, 1–6.

Heisele, B., Kressel, U., and Ritter, W. (1997). Tracking non-rigid, moving objects based on color cluster flow. 1–5.

Liévin, M. and Luthon, F. (2001). A hierarchical segmentation algorithm for face analysis; application to lipreading. 1–5.

Lucchese, L. and Mitra, S. (2001). Color image segmentation: a state-of-the-art survey. 1–15.

Ong, S., Teo, N., Lee, K., Venkatesh, Y., and Cao, D. (2002a). Segmentation of color images using a two-stage self-organizing network. *Image and Vision Computing 20*, 279–289.

Ong, S., Yeo, N., Lee, K., Venkatesh, Y., and D.M.Cao (2002b). Segmentation of color images using a two-stage self-organizing network. *Image and Vision Computing*.

Ozyildiz, E., Krahnstöver, N., and Sharma, R. (2001). Adaptive texture and color segmentation for tracking moving objects. 1–36.

Schaub, H. and Smith, C. (2003). Color snakes for dynamic lighting conditions on mobile manipulation platforms. *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems*, 1–6.

Sobottka, K. and Pitas, I. (1996). Segmentation and tracking of faces in color images. 1–6.

Tremeau, A. and Colantoni, P. (2000). Regions adjacency graph applied to color image segmentation. *IEEE Transactions on image processing*.

The many definitions of computability

Binne Simonides
B.J.Simonides@student.rug.nl

Rowan Rossing
R.Rossing@student.rug.nl

Keywords

Computability theory, Turing machines, Lambda calculus, Recursion theory, Abacus, FEM.

Summary

There are many models of computation introduced to define the notion of computable function. All these characterisations describe the same set set of computable functions. In this paper we study a number of definitions of computability and compare them on a number of properties. All the different systems have their own bad and good properties. Some systems are very useful for educational purposes, and some systems have great theoretical possibilities while others have more practical use.

1 Introduction

The goal of this paper is to study a number of definitions of the notion of computability and to compare them on a number of properties.

In section 2 we give a general introduction to the notion computability. In sections 3 to 7 we describe a some well known definitions of computability. Then, in section 8, we show that these definitions are all equivalent. We conclude the paper with a comparison between the different definitions on some of their properties in section 9.

In this paper we restrict our study to the following characterisations of computability: Turing computable, recursive functions, lambda-definable, abacus computable and FEM-implementable.

2 Computability in general

The informal notion of a computation as a sequence of steps performed according to some kind of recipe goes back to ancient times. Yet for a very long time there was not a precise definition of what it means for a function to be computable, or for a problem to be solvable. This was probably because the absence of such a definition did not cause any difficulties.

In 1900, the mathematician David Hilbert presented a list of 23 problems that he hoped would be solved in the next century. The tenth problem on this list called for a decision procedure for

Diophantine equations or a demonstration that no such procedure exists. Later in that century the problem was solved. It was shown that there exists no such procedure. To show that something *is* computable, is much easier: we only have to give an algorithm that computes it. To show that no computational procedure can solve a certain problem, we need a characterisation of *all* possible computational procedures; we need a formal model of computation.

It was not until the 1930's that such formal models of computation began to arise. At that time a lot of models were developed. Turing provided a notion of mechanical computability, Gödel and Herbrand characterised computability in terms of recursive functions, Church presented the notion of lambda definability and so on. Today, there are even more characterisations like abacus computable and programmability in any number of programming languages.

Although these descriptions of computability are all quite different and were all developed by different people, exactly the same functions can be computed by each model. This is part of the evidence that these definitions capture the intuitive notion of computability. That this is the case is stated in the well-known and widely accepted Church-Turing thesis.

An undecidable problem is one that cannot be solved by any algorithm, even given unbounded time and memory. Many undecidable problems are known. One example is the *Entscheidungsproblem*: given a statement in first-order predicate calculus, decide whether it is universally valid. Another example is the halting problem: given a program and inputs for it, decide whether it will run forever or will eventually halt.

3 Turing computable

Let us first introduce the informal notion of algorithm or "effective method of computation". An effective method is generally assumed to satisfy the following conditions:

1. The method consists of a finite set of simple and precise instructions that are described with a finite number of symbols.
2. The method will always produce the result in a finite number of steps.
3. The method can in principle be carried out by a human being with only paper and pencil.
4. The execution of the method requires no intelligence of the human being except that which is needed to understand and execute the instructions.

The above description is intuitively clear, but certainly not formal definition.

In his 1936 paper *On Computable Numbers, with an Application to the Entscheidungsproblem* Alan Turing tried to formally define the intuitive notion of effective method with the introduction of Turing machines. He showed in that paper that the Entscheidungsproblem could not be solved.

The Turing machine is an abstract model of computer execution and storage; it is mathematically precise definition of algorithm or effective method.

The notion Turing machine can be defined in many different ways, depending on the answers to such questions as the following. 'Is the tape endless in both directions or just in one? How many tapes are there? Do the squares into which the tape is divided have addresses?' And so on. However, the same set of functions is computable regardless of how the details are filled in.

For a nice definition we refer to *Computability and Logic* [11]. We shall only give the formal definition.

The formal definition is: a Turing machine M is a 7-tuple (Q, Σ, Γ, δ, q_0, S_0, F), where:

- Q is a finite set of states,
- Σ is a finite set of symbols, the input alphabet,
- Γ is a finite set of symbols, the tape alphabet,
- δ is the partial transition function ($Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$),
- $q_0 \in Q$ is the initial state,
- $S_0 \in \Gamma$ is a symbol called blank,
- $F \subseteq Q$ is a set of final states.

A function is Turing computable if there exists some Turing machine that computes the function in a finite number of steps. Turing's model can be used for proving that a certain function is uncomputable or that a certain problem is unsolvable. That is in fact why Turing created this model of computation: to prove that the Entscheidungsproblem is unsolvable.

As an example let us consider the busy beaver problem. It can be proved, using Turing machines, that the busy beaver problem is unsolvable. The proof is by *reductio ad absurdum*. An absurdity ($0 \geq 1$) is deduced from the assumption that the busy beaver problem *is* solvable (that there is a Turing machine that computes a certain function p). For a description of the busy beaver problem and the proof we refer to *Computability and Logic* [11].

The halting problem can also be proved unsolvable with the use of Turing machines. Note that this means that there is no Turing machine that computes a solution to the problem. Only if the Church-Turing thesis is true (which is very plausible) there is no effective method in the intuitive sense that solves the problem.

The control unit of a Turing machine can be seen as a finite state machine. This is another benefit of the Turing machine model, because it facilitates the use of the extensive theory on finite state machines in computability theory.

Another good feature of Turing machines is the possibility to simulate a Turing machine on a Turing machine. It is possible to build a Turing machine that accepts the description of another Turing machine on it's tape and computes what that Turing machine would have computed. Such a machine is called a universal Turing machine.

4 Lambda-definable

A few months before Turing had proved the Entscheidungsproblem to be unsolvable (1936) Alonzo Church had proved a similar result in *A Note on the Entscheidungsproblem*. He however used the notions of recursive functions and lambda-definable functions to formally describe effective computability. Lambda-definable functions were introduced by Alonzo Church and Stephen Kleene (Church 1932, 1936a, 1941, Kleene 1935).

However, the lambda calculus was not purposely designed for computability theory. Originally, Church had tried to construct a complete formal system for the foundations of mathematics. When he found out that this system was susceptible to Russel's paradox he separated out the lambda calculus and used it to study computability.

The lambda calculus is a way of defining functions. It is a formal system designed to investigate function definition, function application and recursion.

Formally, we have a countably infinite set of identifiers I. The set of all lambda expressions can then be defined by the following CFG in BNF:

1. `<expr>` → `<identifier>`
2. `<expr>` → (λ `<identifier>` . `<expr>`)
3. `<expr>` → (`<expr>` `<expr>`)

Rules 1 and 2 generate functions. Rule 3 describes the application of a function to an argument. The parenthesis in rules 2 and 3 can be omitted if there is no ambiguity when following these rules:

1. Function application is left-associative.
2. A λ binds to the entire expression following it.

The binding of occurrences of variables is (with induction upon the structure of the lambda expression) defined by the following rules:

1. In an expression of the form V where V is a variable this V is the single free occurrence.
2. In an expression of the form $\lambda\ V\ .\ E$ the free occurrences are the free occurrences in E except those of V. In this case the occurrences of V in E are said to be bound by the λ before V.
3. In an expression of the form $(E\ E')$ the free occurrences are the free occurrences in E and E'.

There is an equivalence relation defined over the set of lambda expressions that captures the intuition that two expressions denote the same function. This equivalence relation is defined by the so-called α-conversion rule and the β-reduction rule. There is also a third rule, the η-conversion rule which expresses the idea of extensionality. We will not define these rules here.

It is possible to define natural numbers and arithmetic operations in lambda calculus. The natural numbers are most commonly defined by the so-called Church integers:

$\mathtt{0} = \lambda f.\lambda x.x$,
$\mathtt{1} = \lambda f.\lambda x.fx$,
etc.

And to give one example of an arithmetic function, addition can be defined as follows:

$\mathtt{PLUS} = \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$.

The computability of a function can be defined with lambda calculus. A function $F : \mathbb{N} \to \mathbb{N}$ is defined to be computable if there exists a lambda expression f such that for every pair of $x, y \in \mathbb{N}$, $F(x) = y$ if and only if the expressions fx and y are equivalent. More detailed information on lambda calculus can be found in [13].

The lambda calculus has had great impact on functional programming languages. In fact a lot of functional programming languages are equivalent to lambda calculus extended with constants and data-types. The programming language LISP is a well-known example. However only its purely functional subset is really equivalent to lambda calculus. Nevertheless lambda calculus seems to have much more practical use than for instance Turing machines.

Lambda calculus produced the historically first problem for which the unsolvability could be proven. There is no algorithm that given two lambda expressions decides if they are equivalent. To proof this Church needed a formal definition of the notion of algorithm. He used a definition via recursive functions for this.

Church's proof first reduces the problem to determining whether a given lambda expression has a normal form. A normal form is an equivalent expression which cannot be reduced any further. Then he assumes that this predicate is computable, and can hence be expressed in lambda calculus. Then he deduces a contradiction from this and thus proves the uncomputability.

5 Recursive functions

The class of recursive functions exists of the basic functions and the functions that can be obtained by the composition, primitive recursion and the minimisation operation on the basic functions.

The basic functions are:

$$z(x) = 0 \text{ (zero function)}$$
$$s(x) = x + 1 \text{ (succesor function)}$$
$$id_i^n(x_1, \ldots, x_i, \ldots, x_n) = x_i \text{ (identity function with index)}$$

The three additional operations are: composition, primitive recursion and minimisation. The following descriptions of these operations are from [11].

If f is a function of m arguments and each of $g_1, \ldots, g_m$ is a function of n arguments, then the function h where

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \text{ (composition)}$$

is the function obtained from $f, g_1, \ldots, g_m$ by composition.

The second sort of operation for generating new recursive functions is primitive recursion.

$$h(x, 0) = f(x), h(x, s(y)) = g(x, y, h(x, y)) \text{ (primitive recursion)}$$

Where h is said to be definable by primitive recursion from the functions f and g.

The third operation is minimisation. Applied to total function f of $n+1$ arguments, the operation of minimisation yields a function h of n arguments as follows.

$$h(x_1,\ldots,x_n) = \begin{cases} \text{the smallest } y \text{ for which } f(x_1,\ldots,x_n,y) = 0, \text{ if any,} \\ \text{undefined if } f(x_1,\ldots,x_n,y) = 0 \text{ for no } y \end{cases} \quad \text{(minimization)}$$

Recursion is best explained by an example which deduces an exponentiation into a multiplication into an addition into a recursive form.

$$\exp(x,y) = x^y$$
$$5^3 = 5^{2+1} = 5^1 * 5^2 = 5 * 5 * 5 * 5^0$$
$$5^3 = 1 + 1 + \ldots\ldots\ldots + 1 \text{ (*125 times*)}$$

We have reduced the exponentiation to an addition problem. The only thing left to do is define the addition operator as a recursive function, we can do this in the following way.

$$x + 0 = x, x + s(y) = s(x + y)$$

As an example we will put the above expressions in primitive recursive form.

$$\text{sum}(x,0) = id_1(x), \text{ where sum} = h$$
$$\text{sum}(x,s(y)) = g(x,y,\text{sum}(x,y)) = s(\text{id}_3(x,y,\text{sum}(x,y)))$$

For the product function we have to put the following equations in primitive recursive form.

$$\text{prod}(x,0), \text{prod}(x,s(y)) = x + \text{prod}(x,y)$$

This can be done in the following manner.

$$\text{prod}(x,0) = z(x), \text{ where prod} = h \text{ and where sum} = g$$
$$\text{prod}(x,s(y)) = g(x,y,\text{prod}(x,y)) = \text{sum}(\text{id}_1(x,y,\text{prod}(x,y)),\text{id}_3(x,y,\text{prod}(x,y)))$$

A benefit of the recursive functions is that it has a very small instruction set. This fact can be used to develop extremely reliable systems, because we can test them thoroughly. A small instruction set makes hardware errors easy to discover and recover.

The downside is it takes a lot of instructions even for simple computations; i.e. the computation $2+5$ needs `sssssss0` $= 7$ instructions in a recursive functions system while it would require one instruction on a modern computer.

Of course we can optimise by building special hardware for the basic functions of recursion, and we can add a few extra functions (like addition). This does make the instruction set more complex so this is a trade-off we have to take into account.

6 Abacus computable

The first abacus machine was developed 5000 years ago, and was used by Greek and Chinese tradesmen. The word abacus came from the Greek word abakos (for a board or a tablet) which, in turn, was probably derived from the Hebrew abaq which means dust. The sand surface which was used earlier for writing had evolved into a board with lines. The modern abacus is a wooden frame fitted with rigid wires on which counters made of wood or plastic can slide.

The basic principle of the abacus is quite simple, it uses an infinite number of boxes (registers) which can each contain an infinite amount of stones. The two basic operations are depicted in figure 1.

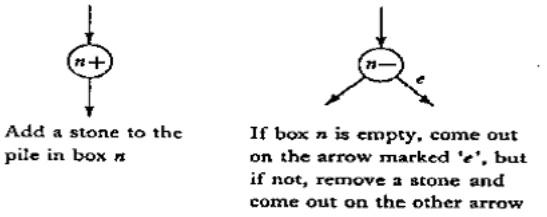


Figure 1: Elementary operations in abacus.

These basic operations can be used to make an addition or a multiplication see figure 2. In the addition registers m and n are added and the result is put in register n, register p is initially empty. The multiplication uses the same principle where box m is used as a counter.

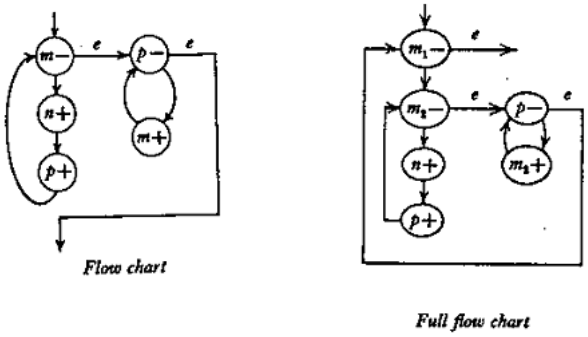


Figure 2: Addition on the left, multiplication on the right in abacus.

The above example can easily be extended to exponentiation, this is done in [11].

7 FEM-implementable

FEM is a relatively new abstract functional programming language. It is developed by Jan Terlouw on the base what already had been done in literature concerning combinatory logic and

lambda-calculus. It is specifically designed for educational purposes and in particular for courses on functional programming and the semantics of formal languages at the University of Groningen. The name FEM is inferred from Functional programming and sEMantics of formal languages. A detailed description of FEM can be found in Terlouw's syllabus [12]. More information on the underlying theories can be found in [13].

This language is a rather abstract functional language which makes it useful for demonstrating and studying the diverse aspects of functional programming in a pure form. The language is very powerful and although it doesn't look very high level it is surprisingly user-friendly. To keep things compact we will give here a somewhat informal definition of a FEM-program. For the formal definition we refer to Terlouw's paper.

A FEM-term is constructed from constants and variables. There is an enumerable infinite set of constants $\mathbf{Con} = \{\mathbf{s}, \mathbf{p}, \mathbf{t}, \overline{0}, \overline{1}, \overline{2}, \ldots\}$. There is also an enumerable infinite set of variables with $\mathbf{Var} \cap \mathbf{Con} = \emptyset$. The set of all FEM-terms **Ter** is inductively defined as:

1. $A \in \mathbf{Con} \rightarrow A \in \mathbf{Ter}$
2. $A \in \mathbf{Var} \rightarrow A \in \mathbf{Ter}$
3. $A \in \mathbf{Ter} \wedge B \in \mathbf{Ter} \rightarrow (AB) \in \mathbf{Ter}$

The intuitive semantic of FEM-terms is:

1. The constants $\overline{0}, \overline{1}, \overline{2}, \ldots$ represent the natural numbers; the constants **s** and **p** represent the successor and predecessor functions on the set of natural numbers. The constant **t** represents a test for zero branching operation. The term $\mathbf{t}ABC$ can be read as "if $A = 0$ then B else C".
2. Variables have values defined by FEM-declarations.
3. In a composed FEM-term (AB) the function A is applied on argument B. The value of (AB) is the result of the application. The parenthesis can be omitted; application is left-associative.

A FEM-declaration is an expression of the form $x\vec{y} = B$ where x is a variable, $\vec{y}$ a possibly empty sequence of independent variables different from x and B a FEM-term. The variable x is called the *declared variable* and the variables from $\vec{y}$ are called the *formal parameters* of the declaration. With a declaration - possibly in combination with other declarations - the value of a variable is defined. Declarations can be recursive.

A *context* is finite sequence of FEM-declarations. A FEM-program is a pair (Γ, A) where Γ is a context and A a *FEM-term* (the *main term* of the program). The value of a FEM-program (Γ, A) is the value of it's main term A in which every variable gets a value defined in Γ.

Now we can define when a function is FEM-implementable. Let $k \in \mathbb{N} \setminus \{0\}$ and $f : \mathbb{N}^k \rightarrow \mathbb{N}$. The k-dimensional numerical function f is FEM-implementable if there is a FEM-program (Γ, A) with the following property: for all $n_1, \ldots, n_k \in \mathbb{N}$ the FEM-program $(\Gamma, A\overline{n_1} \cdots \overline{n_k})$ has value $f(n_1, \ldots, n_k)$.

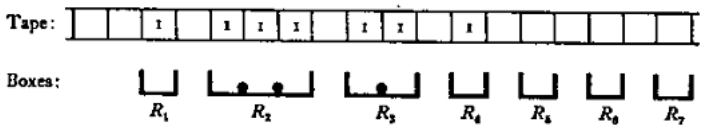


Figure 3: Correspondence between boxes and portions of the tape.

8 Equivalence proofs and the Church-Turing thesis

Now we have given the philosophy and characterisations of the following definitions of computability: Turing computable, lambda-definable, recursive functions, abacus computable and FEM-implementable. These definitions are just a few (although some very important) of the many definitions of computability.

All these characterisations define their own formal model of computation. With that model they describe a set of functions computable with that model. The interesting fact is: all these models describe the same set of computable functions.

It was shown by Church and Kleene in 1936 that the formalisms of the lambda-definable functions and the recursive functions describe the same set of functions. Shortly after that Turing showed that this set of functions is equal to set of Turing computable functions. This led to the Church-Turing thesis which states that Turing machines indeed capture the informal notion of effective or mechanical method in logic and mathematics. The many systems that were introduced after that were all shown to compute the same functions as Turing machines. These systems are called Turing-complete.

The thesis can not be proven, so it does not have the status of a theorem. In theory the thesis could be disproved with an algorithm that is generally accepted as an effective method but cannot be performed on a Turing machine. But since all the different attempts at formalising the notion of algorithm have yielded the same set of computable function, the thesis is now generally accepted to be true. In fact, it is so widely accepted that modern mathematicians often use the term Turing computable instead of the undefined terminology.

Now we briefly discuss the nature of the equivalence proofs.

It can be proved that abacus computable functions are Turing computable ($A \subseteq T$) by choosing a standard format for both computations and presenting a method for converting the flow graph of an abacus A into a Turing machine which computes the same functions. Figure 3 shows an example for the format one can choose.

If we want to add a stone to a box (see left figure 1) we can do the following:

1. Start at standard position (leftmost position on the tape).
2. Find register n.
3. Add a '1' on the tape.
4. If there are filled registers on the right, then move them all 1 place to the right, if not go back to the standard position.

For emptying a register n (see right figure 1) we do the same except we have to move all the registers to the left and we need an exit in case the register is empty. This is done in detail in [11].

To prove that the recursive functions are abacus computable ($R \subseteq A$) one first has to show the abacus computation for the basic functions $z(x), s(x), \mathrm{id}(x)$. After that one must show how given programs which compute functions $f, g_1, \ldots, g_m$, how they can be arranged into a program which computes their composition $h = \mathrm{Cn}[f, g_1, \ldots, g_m]$. And, given programs which compute functions f and g, how they can be arranged into a program which computes the function h obtained from them by primitive recursion, $h = \mathrm{Pr}[f, g]$. This proof is also given in [11].

It can also be proved that Turing computable functions are recursive ($T \subseteq R$). This is also done in [11]. We now have $R \subseteq A \subseteq T \subseteq R$ which means that $R = A = T$. Similarly it can also be proved that the set lambda-definable functions L and the set of FEM-implementable functions F are also equal to R, A and T. Details on this can be found in [13]. Thus it can be proved that $R = A = T = L = F$; all these systems of computation describe the same set of computable functions.

9 Conclusion: the comparison chart

All characterisations describe the same set of computable functions. So one could say that the expression power of all models is the same. However these models also have their own specific properties and some models could be more suitable for certain purposes than others.

Therefore we will introduce a number of properties on which we shall compare the different systems. For completeness we have also included expression power. The properties are: expression power, theoretical power, practical power, educational power, compactness of algorithms, compactness of theory, accessibility.

Theoretical power expresses the usefulness for studying computability theory, for instance how well it can be used to prove the uncomputability of functions. Practical power expresses the systems usefulness in practical computer science, such as it's influence on programming languages. Educational power indicates it's use in education. How compact algorithms can be described with certain system is expressed in the compactness of algorithms. Compactness of theory expresses the compactness of the total definition of computability with that characterisation. Accessibility means how accessible the characterisation is.

We have made a classification based on our study of different models of computation. We have rated the properties of the systems with 1-3 stars, where three stars is best. Table 1 below shows our results.

Motivation

- The expression power of all systems is the same ($\star\star\star$), because they are all equivalent.
- The theoretical power of Turing machines and recursive functions is high ($\star\star\star$), because these two systems are used a lot for proving the uncomputability of functions. The theoretical power of lambda calculus is slightly less ($\star\star$), because it is more focused on functional programming. Abacus and FEM have relatively low theoretical power ($\star$); they are not used in complicated theoretical applications.

Table 1: The comparison chart

	Turing machines	lambda calculus	recursive functions	abacus	FEM
Expression power	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆
Theoretical power	⋆⋆⋆	⋆⋆	⋆⋆⋆	⋆	⋆
Practical power	⋆⋆	⋆⋆⋆	⋆⋆	⋆⋆	⋆⋆
Educational power	⋆⋆⋆	⋆⋆	⋆⋆	⋆	⋆⋆⋆
Compactness of algorithms	⋆	⋆⋆⋆	⋆⋆⋆	⋆	⋆⋆⋆
Compactness of theory	⋆⋆	⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆
Accessibility	⋆⋆⋆	⋆	⋆	⋆⋆⋆	⋆⋆⋆

- Lambda calculus has very high (⋆⋆⋆) practical use,because of its great influence on functional programming languages. The other systems have slightly less practical use (⋆⋆); they can all be simulated on modern computers though.
- Turing machines have a lot of educational power (⋆⋆⋆), because they are easy to comprehend (we can easily make a mental picture of them). FEM too has great educational power (⋆⋆⋆), because many aspects of functional programming can be studied with it in a pure way and it still is reasonably user friendly. Lambda calculus and recursive functions have less educational power because of their complexity; it takes some time and dedication to get used to their formalisms. An abacus is useful for children to learn how to count (⋆).
- Lambda calculus, recursive functions and FEM have all three very compact notations (⋆⋆⋆). Turing machines and abacus both have very large notations (flow-charts, finite state machines and other diagrams) (⋆).
- The theory of abacus is by far the most compact (⋆⋆⋆); it only has two basic operations. The other theories are also compact (⋆⋆), but they need a few more operations/instructions.
- The accessibility of Turing machines and abacus is very good, because one can easily make a mental picture of the workings of these systems. FEM is also easy to understand, because it is more high level than lambda calculus and recursive functions (⋆⋆⋆). Lambda calculus and recursive functions are more difficult, because they have fewer but more complex operations (⋆).

Looking at this chart, FEM-implementable appears to be the most all-round definition of computability. However, we understand that there are many possible arguments for a different distribution of the stars.

References

[1] Computability and Incompleteness, Lecture Notes, Jeremy Avigad (July 26, 2002)

[2] Wikipedia, the free encyclopedia, `http://www.wikipedia.org/`

[3] Turing, A., On Computable Numbers, With an Application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Series 2, Volume 42, 1936; reprinted in

M. David (ed.), The Undecidable, Hewlett, NY: Raven Press, 1965; online: http://www.abelard.org/turpap2/tp2-ie.asp

[4] Church, A., A note on the Entscheidungsproblem, Journal of Symbolic logic, 1 (1930), 40 - 41.

[5] Church, A. 1932. A set of Postulates for the Foundation of Logic. Annals of Mathematics, second series, 33, 346-366.

[6] Kleene, S.C. 1936, Lambda-Definability and Recursiveness. Duke Mathematical Journal, 2, 340-353.

[7] Kleene, S.C., A theory of positive integers in formal logic, American Journal of Mathematics, 57 (1935), pp 153 - 173 and 219 - 244.

[8] Church, A., An unsolvable problem of elementary number theory, American Journal of Mathematics, 58 (1936), pp 345 - 363.

[9] Gödel, K. 1934. On Undecidable Propositions of Formal Mathematical Systems. Lecture notes taken by Kleene and Rosser at the Institute for Advanced Study. Reprinted in Davis, M. (ed.) 1965. The Undecidable. New York: Raven.

[10] Herbrand, J. 1932. Sur la non-contradiction de l'arithmetique. Journal fur die reine und angewandte Mathematik, 166, 1-8.

[11] Boolos, G.S. and Jeffrey, R.C., Computability and Logic, Cambridge Universtiy Press, 3rd edition, reprinted 1991.

[12] Terlouw, J., De abstracte functionele programmeertaal FEM, RuG-Informatica (September 1, 2003).

[13] H.P. Barendregt, The Lambda Calculus, Its Syntax and Semantics, Elsevier Science Publishers B.V., Amsterdam, revised edition, 1984.

ERP: Does it live up to its promises?

An overview of implementation strategies and system disadvantages.

Daniel Neeteson, Auke Schotanus

CS/SSE department
University of Groningen
January 2004

Abstract

Enterprise Resource Planning (ERP) is a management tool which is used to link several departments within a company. This paper presents a layout of the benefits and problems which companies encounter when implementing their enterprise systems. Strategies are explored to maximise benefits from these enterprise systems in different ways. Thorough analysis of both ERP software and organisational processes, in this article called system exploration, has proven to be the most successful implementation strategy. However, there are some disadvantages in implementing ERP. ERP does not cover all business processes in a satisfactory manner, and many organisations have implemented human resource tools (HR-tools) in addition to an existing ERP implementation. Another disadvantage is the limited number of modules that are used by an organisation. This causes unused modules costing time and money. Some organisations have expected too much from ERP, and careful analysis is needed to achieve a return of investment within a few years.

1.0 Introduction

1.0.1 What is ERP?

ERP is an acronym for Enterprise Resource Planning. The definition of ERP is; the practice of consolidating an enterprise's planning, manufacturing, sales and marketing efforts into one management system (CIO, 2003). The first of the three words encapsulates the main concept of ERP. ERP focuses on trying to integrate departments and functions throughout a company. It attempts to integrate everything into a single system that can serve every department's and function's needs (The Simple ERP Site, 2003). The idea of ERP started in the 1960s. Back then there really wasn't a name classification of this concept. Its concept was to integrate all departments and functions to increase revenues and strengthen the business. In 1972, five managers who came from IBM started a business in ERP, which was known as SAP (The Simple ERP Site, 2003). The core strength of accounting and ERP software is in financial reporting, inventory management, purchasing and order processing. As with any large software application, ERP applications have been expanding their offering by adding more features. Some of these new features include: supply chain management (SCM), customer relationship management (CRM) and more recently professional services automation (PSA) (Melik, 2002).

1.0.2 Overview and Research Question

In this article we will give you an overview of a model for maximising benefits from enterprise systems, and the strategies involved. After that we will give you a short overview of the disadvantages of using enterprise systems. The research question is: What is the best

way to implement ERP considering the advantages, disadvantages and feasibility of such an implementation?

2.0 How can organisations maximise benefit from enterprise systems?

Many organisations have spent millions of dollars on packaged enterprise application software from vendors such as SAP, Oracle, PeopleSoft, i2 and Arriba. Worldwide sales are estimated at around US$ 21.4 billion in 2004 (E-Pay News Statistics, 2004). This amount does not include additional implementation costs of hardware, networks and consultancy, so the number may be inflated by a factor three to five (Smith, 1999). Even this is not enough: think of all the hidden costs to companies of the internal resources deployed on projects and the consequent loss of focus on the business (Smith, 1999). Considering these high costs it is important for businesses to ask themselves how to maximise the benefits of these huge investments. This part of the article will analyze different ways of enterprise system use in order to make clear what the best practice is and why some companies fail to benefit from ERP.

In this article, the term *packaged enterprise application software* (PEAS) refers to the packaged software itself, and the term *enterprise systems* (ES) refers to the combination of people, organisational processes, hardware, telecommunications networks, and software that use PEAS (Shang, 2002).

Packaged enterprise application software products are highly flexible. They contain solutions to the needs of many of the vendor's customers. Because these customers are from a very broad range of industries, it is very hard for a specific organisation to find the right combination of processes for its own changing needs and to implement those processes in its own organisation (Shang, 2002). The challenge for an organisation implementing ERP is to achieve an on-going fit between the evolving capability of the software and the changing needs of the organisation. Of course, it is achieving this on-going fit what makes the implementation of PEAS so hard and costly. The complexity of PEAS is such that, even when the implementation is done very well, it is not likely that the first implementation will fit the organisation's processes very well. Understanding the potential benefits of enterprise systems takes time for the organisation, and the organisation's needs will change because the organisation changes itself (strategy, markets, reorganisations). Meanwhile, new versions of PEAS are delivered and new technologies are introduced by the vendors. Vendor firms themselves could even merge or go bankrupt. To achieve on-going fit, understanding both the organisation's processes and the capabilities of the PEAS is of great importance for managers.

PEAS are a semi-finished product that user organisations must tailor to their business needs (Shang, 2002). Tailoring the software using parameters (e.g. base currency) provided by the vendor is usually called *configuration* (Bancroft et al., 1998). The functionality of PEAS can also be changed by altering the actual program code so that it fits the processes of the organisations in a better way. This is usually called *customisation* (Shang, 2002). Customisation is usually not recommended by vendors, because it imposes the risks of software development itself and the possibility of having to re-customise new releases of the software. Customisation can be avoided by changing business processes to match those supported by the software. However, changing organisational processes is difficult and processes supported by the software may not be in the interests of the organisation.

The challenge for teams implementing PEAS is to find an optimum between configuration, customisation and process change. This optimum should not be ad hoc, but it must last over time, so finding out what processes are important now and in the future, and a thorough understanding of the ES software is crucial. This problem can be compared to the partitioning problem in hardware software co-design (de Micheli, 1999), where the problem is to find an optimum between the flexibility of software and the speed of hardware.

2.1 A Model for maximising benefits from ES

In this article we will use the model as depicted in the article by Shang. This model is specific for ES projects and leaves out all factors that could influence any other IT project. At the left, four "Distinctive Characteristics of PEAS" are shown. Each of these characteristics needs to be managed, because it can be a source of value or of problems for the implementing organisation. Configuration activities are done at the left of the model. The goal of the activities depicted in the middle of the figure is to achieve an on-going fit between the evolving capabilities of the software and the changing needs of the organisation. Changes at

Figure 1: Maximizing Benefits from Enterprise Systems

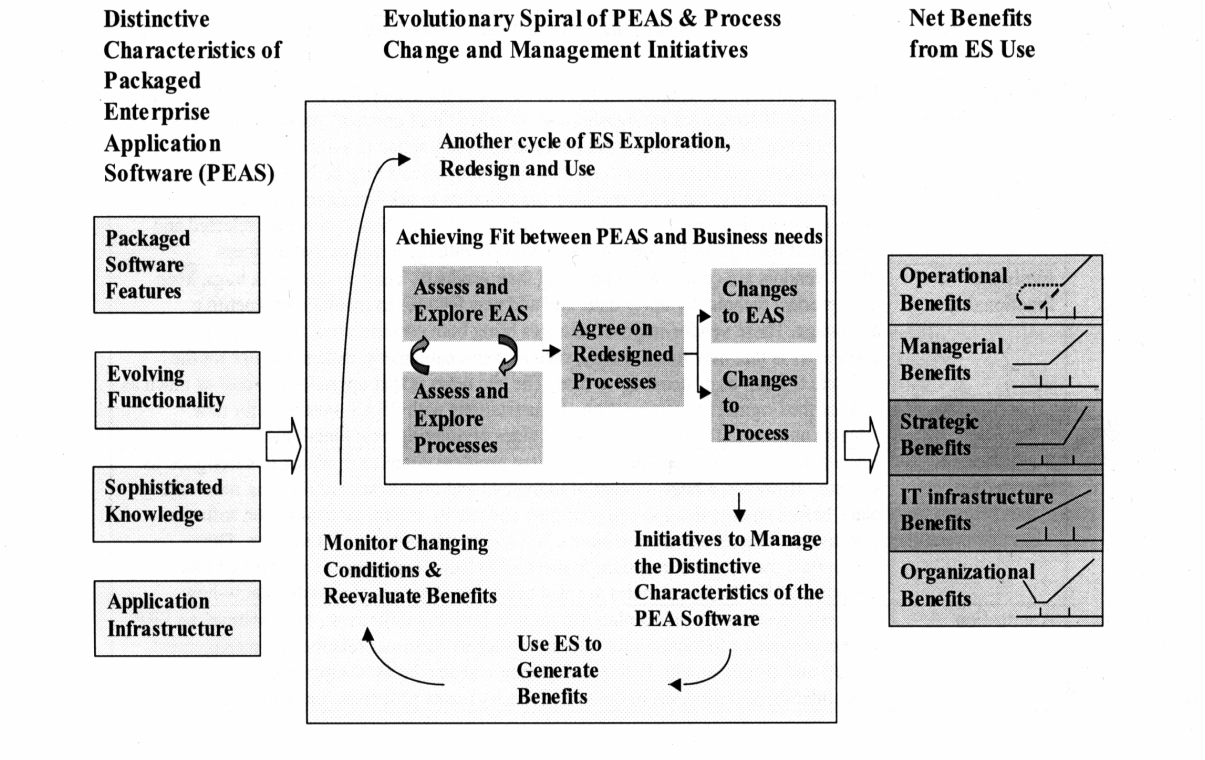


Figure 1: Taken from Shang & Seddon, p9, 2002

the left will cause new cycles in the middle of the figure. At the right of the model, you can see five types of benefits from ES use. The graphs are based on the results of four Australian companies that were investigated by Shang & Seddon with a time axis range of 3 years. The fact that benefits were higher in year three is important because by year three all the companies had adopted an implementation strategy Shang & Seddon call *System Exploration*, which will be discussed later in this article.

2.2 The four strategies for achieving fit

As stated before, achieving fit between the organisation and enterprise systems is a matter of changing the software to the organisation's needs or changing the organisation's processes to the functionality of the acquired software. Combining these two activities, there are four strategies for achieving fit, depending on the presence or absence of preparedness to do these activities, which will be discussed in the following subsections.

2.2.1 Process replication

Process replication is done when the preparedness of both changing PEAS and changing organisational processes is low. Process replication means using the PEAS to duplicate or automate existing business processes. This strategy is used by companies that want to reduce costs and risks of software and organisational changes (Shang, 2002). Furthermore, because of the boom market at the end of the 20th century, it was in the interests of implementation partners to move to different clients quickly, without spending time to change things (Smith, 1999). Moreover, because the scale and complexity of implementation were underestimated, many firms chose the simple route with this strategy (Smith, 1999).

2.2.2 Process Modification & Enhancement

When the organisation makes changes to organisational processes in order to adapt these processes to be configurable within PEAS, it is called *process modification & enhancement*. The main risk of this strategy is that the software may dominate the way they conduct their business.

2.2.3 Software Modification & Enhancement

When the organisation makes changes to the PEAS by configuring and customising it to fit the existing processes as closely as possible, it is called *software modification & enhancement*. Vendors of PEAS usually do not recommend this strategy, because it will probably mean reconfiguring and re-customisation when a new release is installed. However, some organisations believe their most important processes are so unique that they have to change the software to fit these special processes.

2.2.4 System Exploration

When an organisation explores all opportunities for better process performance by being prepared to make changes to both business processes and the PEAS, it is called *system exploration*. This strategy needs process managers that have a very good understanding of both the PEAS and the organisational processes.

2.3 Evaluation of the different strategies

All these strategies have pros and cons. Process replication caused early automation benefits (speed and cost), but previous process problems are not assessed and therefore persist (Shang, 2002; Smith, 1999). Process modification revolutionised business processes, but holds the risks of data errors, work mistakes and user resistance (Shang, 2002). Software modification made changeover to the new system relatively simple, but the costs are higher because of software maintenance and upgrades. Finally, system exploration produced the most benefits, but process owners have such a complex task that they suffered from work overload. All companies investigated in the study by Shang & Seddon had changed their strategy to system exploration by the third year of ES use, and rise of the benefits in the third year is compelling evidence that system exploration is the most appropriate strategy. The model proposed by

Shang & Seddon depicts the evolutionary-life-cycle as used by the process managers of the companies.

2.4 Institutionalisation of process-improvement is necessary for successfully implementing PEAS

In order to use system exploration in the best way, it is important to form *Centers of Excellence* to institutionalise processes for achieving on-going fit (Shang, 2002). Both Shang & Seddon and Mark Smith strongly recommend these teams to continue their work after the implementation has gone live. It is important to articulate the benefits clearly so that they can be measured for re-evaluation. Mark Smith emphasises the importance of the development of detailed business cases, something which is supported by Shang & Seddon by stressing the need for process managers to understand their organisation processes in a thorough way. It is this long-term thorough analysis which produces the greatest benefits when implementing PEAS.

3.0 Disadvantages of ERP implementation

The use of ERP has grown considerably at the end of the 20th century. Many companies have implemented ERP just because everybody else was doing it, not considering the costs and return of investment (Smith, 1999). This has led to many poor implementations within organisations. These organisations did not achieve the on-going fit while spending huge amounts of money.

Enterprise applications show strength in financial reporting, inventory management, purchasing and order processing (Melik, 2002). ERP lacks support for project and people management. Enterprise Resource Planning has more to do with accounting registering than with planning. Sophisticated time scheduling and project planning are not supported by ERP. Because of these shortcomings, many organisations have installed separate software packages that support *human resource management* (HRM). The goal of an ERP system is to integrate different organisational activities in one single system. This integration has clearly not been achieved when companies acquire additional software packages (Melik, 2002).

Another disadvantage of ERP is the size of the product as delivered by the vendor. Most companies use only a limited number of modules of the system. This leaves many unused modules, that also have to be configured every time a new version is installed. This leads to an unnecessary rise in implementation costs (Melik, 2002).

Considering the title of this article, ERP does not live up to its promises. It was presented as a panacea, but turned out to be costly and difficult to implement for a lot of companies. It does not cover all expected areas of business processes. However, a thoroughly and well-prepared implementation does pay off in a few years.

4.0 Conclusion

Nowadays many organisations invest heavily in PEAS. These packages are highly costly and incredibly complex. It takes a lot of time and effort and money to implement these systems. The research question was: What is the best way to implement ERP considering the advantages, disadvantages and feasibility of such an implementation? We have shown a model from Shang and Seddon which shows an evolutionary spiral for achieving maximum benefits. Achieving fit between organisations and enterprise system is an ongoing process. For achieving this fit we have pointed out four strategies:

- 1. Process replication
- 2. Process modification and enhancement
- 3. Software modification and enhancement
- 4. System exploration

We have shown that system exploration is the most effective strategy to adapt the enterprise system to an organisation. Organisations must realise what ERP can and cannot do, so that it can meet its expectations. An important cause of implementation failure is short-term decision making which is always done under severe time pressure. As we said before: it is this long-term thorough analysis which produces the largest benefits when implementing PEAS.

5.0 References

BANCROFT, N.H., SEIP, H. & SPREGEL, A., Implementing SAP R/3*, Greenwich, CT: Manning Publications Inc.*, 1998.

EPAYNEWS STATISTICS: http://www.epaynews.com/statistics/ecappstats.html

MICHELI, G. DE, Hardware Software Co-Design. *Proceedings of the IEEE*, 1997:85(3), p349-364

SHANG, S. & SEDDON, P.B., 2002, Maximising benefits from enterprise systems.

SMITH, M. Realising the benefits from investment in ERP. *Management Accounting*, november 1999, p34.

CIO.COM, Executive summaries: Enterprise Resource Planning, http://www.cio.com/summaries/enterprise/erp/index.html, September 8 2003.

THE SIMPLE ERP SITE, http://www.du.edu/~atanner/index.htm.

MELIK, R., 2002, Competitive disadvantages of ERP based project and human capital management. http://www.tenroxpsa.com/en/downloads/whitepapers/ERP/index.htm.

Appendix A

This appendix contains the review forms that are used during the review process of StudColl.

Review form (to be sent to programme committee and authors)

Paper no.	(use the set number here)
Author(s)	
title	

Please note that this paper is sent to you only for the purpose of reviewing. It is to remain confidential until it is actually published in the conference proceedings. You should not pass it on or disclose it to anyone else. Delegation of the reviewing to someone else is not allowed.

Papers should be regarded as survey papers.

Rating is done according to the following scheme:

rating	meaning		
1	No	Very bad	Completely rewrite
2	No, but …	Bad	Adjust
3	Maybe	Intermediate	Neutral
4	Yes, but …	Good	Accept
5	Yes	Very good	Strongly accept

Short summary of the paper:

Questions	Rating
1. Does the title reflect the contents of the paper?	
remarks:	
2. Is the subject matter clearly within the scope of the StudColl conference? (i.e., interesting and understandable for this particular audience)	
remarks:	
3. Is the paper clearly focused?	
remarks:	
4. Is the central thesis of the paper well founded and well discussed?	
remarks:	
5. Is the presentation (layout, style, grammar, spelling) acceptable?	
remarks:	
6. Should the paper be accepted for the studColl conference?	
☐ accept with minor changes ☐ accept with major changes ☐ completely rewrite **remarks:**	

Please motivate all answers (i.e., give ratings and fill in the remarks per question).

Points in favour or against (sent to the author(s))

The remarks stated here will be sent to the author who may use them to improve the paper.

Detailed textual remarks to be sent to the author

The remarks stated here will be sent to the author who may use them to improve the paper.

Review form (to be sent to the programme committee[1] only)

Paper no.	(use the set number here)
Author(s)	
title	

Remarks for the programme committee

The remarks stated here will only be used during the selection process. They will not be communicated to anybody outside the programme committee.

[1] Jan Terlouw and Rein Smedinga

Re-Review form (to be sent to programme committee[1] only)

Paper no.	(use the set number here)
Author(s)	
Title	
Reviewer	

Please note that this paper is sent to you only for the purpose of reviewing. It is to remain confidential until it is actually published in the conference proceedings. You should not pass it on or disclose it to anyone else. Delegation of the reviewing to someone else is not allowed.

Papers should be regarded as survey papers.

Rating is done according to the following scheme:

rating	meaning		
1	No	Very bad	Strongly reject
2	No, but …	Bad	Reject
3	Maybe	Intermediate	Neutral
4	Yes, but …	Good	Accept
5	Yes	Very good	Strongly accept

For this re-review please use the *second* version of the paper and take into account the way the paper has been improved according to the comments of *all* the reviewers (including your own comments) in the first round of the reviewing process.

In the following table, please give *the final ratings* for each of the items. Use the remark-field that comes with the last two questions to motivate your final ratings.

Short summary of the paper:

[1] Jan Terlouw and Rein Smedinga

Questions	Rating
1. Does the title reflect the contents of the paper?	
2. Is the subject matter clearly within the scope of the StudColl conference? (i.e., interesting and understandable for this particular audience)	
3. Is the paper clearly focused?	
4. Is the central thesis of the paper well founded and well discussed?	
5. Is the presentation (layout, style, grammar, spelling) acceptable?	
6. Has the paper improved in comparison with the first version? In what way?	
remarks:	
7. Should the paper be accepted for the studColl conference?[2]	
☐ accept ☐ reject **remarks:**	

Please motivate all answers (i.e., give ratings and fill in the remarks per question).

[2] papers that are accepted will be published in the proceedings. All papers (both accepted and rejected) will be presented during StudColl.