A Discourse on Complexity of Process Models
(Survey Paper)

J. Cardoso[1], J. Mendling[2], G. Neumann[2], and H.A. Reijers[3]

[1] University of Madeira
9000-390 Funchal, Portugal
jcardoso@uma.pt
[2] Vienna University of Economics and Business Administration
Augasse 2-6, 1090 Vienna, Austria
{jan.mendling, neumann}@wu-wien.ac.at
[3] Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
h.a.reijers@tm.tue.nl

Abstract. Complexity has undesirable effects on, among others, the correctness, maintainability, and understandability of business process models. Yet, measuring complexity of business process models is a rather new area of research with only a small number of contributions. In this paper, we survey findings from neighboring disciplines on how complexity can be measured. In particular, we gather insight from software engineering, cognitive science, and graph theory, and discuss in how far analogous metrics can be defined on business process models.

1 Introduction

Since business process management has become an accepted concept for the implementation and integration of large-scale information systems, there is an increasing need for insight into how errors can be avoided, how maintenance can be facilitated, or how the quality of the processes can be improved. In this context, there is some evidence that complexity is a determinant of error probability of a business process [18]. As process complexity and its measurement is a rather new field in business process management, there is only a limited understanding of how far existing knowledge of complexity e.g. for the software engineering domain can be adopted.

The complexity of a software program comes in three 'flavors': computational complexity, psychological complexity, and representational complexity [26]. The most important is psychological complexity, which encompasses programmer characteristics, product/documentation complexity and problem complexity. Obviously, the latter aspect, the complexity of the problem itself, cannot be controlled in developing software. It is therefore frequently dismissed from consideration in the software engineering literature. It seems sensible to do the same for analyzing the complexity of process models. However, the issue remains that complex processes will require more complex process models. Therefore, for

J. Eder, S. Dustdar et al. (Eds.): BPM 2006 Workshops, LNCS 4103, pp. 117–128, 2006.

the development of process model complexity it seems worthwhile to evaluate complexity measures as *relative* to the underlying process complexity.

Existing theoretical approaches to formulate 'complexity metrics' for software include the use of information theory from signal processing (e.g. [10]) and communication theory (e.g. [24]), as well as approaches based on analogues with graph theory (e.g. [15]) and lattice theory (e.g. [11]). Approaches taking the cognitive sciences as starting point have resulted, for example, in Bastani's complexity model [4]. An overview of some 50 different software complexity metrics is provided in Table 1 in [5].

In this paper, we contribute to a better understanding of business process model complexity. In particular, we provide a theoretical survey of complexity considerations and metrics in the fields of software engineering, cognitive science, and graph theory and we relate them to business process modelling. A further empirical investigation might ultimately lead to establishing a complexity theory of business process models. Following this line of argumentation, the rest of the paper is structured as follows. Section 2 discusses complexity metrics for software and their applicability for business process models. After a general introduction to the discipline, we define analogous metrics to the Line-of-Code, McCabe's Cyclomatic Complexity called Control-Flow Complexity, Halstead Complexity Metric, and Information Flow Complexity as defined by Henry and Kafura. Section 3 relates findings from cognitive science to measuring complexity in software engineering. In Section 4 graph theoretical measures are considered as potential complexity metrics for business process models. Section 5 closes the paper and gives an outlook on future research with a focus on how the process complexity metrics can be validated.

2 Complexity in Business Processes

2.1 Software Metrics

Over the last 30 years many measures have been proposed by researchers to analyze software complexity, understandability, and maintenance. Metrics were designed to analyze software such as imperative, procedural, and object-oriented programs. Software measurement is concerned with deriving a numeric value for an attribute of a software product, i.e. a measurement is a mapping from the empirical world to the formal world. From the several software metrics available we are particularly interested in studying complexity metrics and find out how they can be used to evaluate the complexity of business processes.

Software metrics are often used to give a quantitative indication of a program's complexity. However, it is not to be confused with computational complexity measures (cf. $O(n)$-Notation), whose aim is to compare the performance of algorithms. Software metrics have been found to be useful in reducing software maintenance costs by assigning a numeric value to reflect the ease or difficulty with which a program module may be understood.

There are hundreds of software complexity measures that have been described and published by many researchers. For example, the most basic complexity

measure, the number of lines of code (LOC), simply counts the lines of executable code, data declarations, comments, and so on. While this measure is extremely simple, it has been shown to be very useful and correlates well with the number of errors in programs.

2.2 The Analogy Between Software and Business Processes

While traditional software metrics were designed to be applied to programs written in languages such as C++, Java, FORTRAN, etc, we believe that they can be revised and adapted to analyze and study business processes characteristics, such as complexity, understandability, and maintenance. We based our intuition on the fact that there is a strong analogy between programs and business processes, as argued before in e.g. [23,9]. Business process languages aim to enable programming in the large. The concepts of programming in the large and programming in the small distinguish between two aspects of writing the type of long-running asynchronous processes that one typically sees in business processes. Programming in the large emphasis is on partitioning the work into modules whose interactions are precisely specified and can refer to programming code that represents the high-level state transition logic of a business process (typically using splits and joins). This state transition logic included information such as when to wait for messages from incoming transitions, when to activate outgoing transitions, and when to compensate for failed activities, etc.

A business process, possibly modeled with a language such as BPEL [2], can be seen as a traditional software program that has been partitioned into modules or functions (i.e. activities) that take in a group of inputs and provide some output. Module interactions are precisely specified using predefine logic operators such as sequence, XOR-splits, OR-splits, and AND-splits. There is a mapping that can be established between software programs constructs and business processes. Functions, procedures, or modules are mapped to activities. Two sequential software statements (i.e. instructions or functions) can be mapped to two sequential process activities. A 'switch' statement can be mapped to a XOR-split. In programs, threads can be used to model concurrency and can be mapped to AND-splits. Finally, the conditional creation of threads using a sequence of 'if-then' statements can be mapped to an OR-split.

2.3 Business Process Metrics

We believe that the future for process metrics lies in using relatively simple metrics to build tools that will assist process analysts and designer in making design decisions. Furthermore, because business processes are a high-level notion made up of many different elements (splits, joins, resources, data, activities, etc.), there can never be a single measure of process complexity. The same conclusion has been reached in software engineering. Nagappan et al. [20] point out that there is no single set of complexity metrics that could act as a universally best defect predictor for software programs. For this reason several process metrics can be designed to analyze business processes. For example, Cardoso [8] identifies

four main types of complexity metrics for processes: activity complexity, control-flow complexity, data-flow complexity, and resource complexity.

The following sections describe several approaches to adapt known software metrics proposed by researches worldwide to business processes analysis. Having established that there is a mapping from traditional programming languages and business processes; we will study and adapt some of the most well known and widely used source code metric, i.e. number of lines of code (`LOC`) [13], McCabe cyclomatic complexity [15,16], Halstead's software science measures [10], and Henry and Kafura [12] information flow metric.

2.4 Adapting the `LOC` Metric

One of the earliest and fundamental measures based on the analysis of software code is based on the basic count of the number of Lines of Code (`LOC`) of a program. Despite being widely criticized as a measure of complexity, it continues to have widespread popularity mainly due to its simplicity [3]. The basis of the `LOC` measure is that program length can be used as a predictor of program characteristics such as errors occurrences, reliability, and ease of maintenance.

If we view a process activity as a statement of a software program, we can derive a very simple metric (metric *M1*) that merely counts the number of activities (*NOA*) in a business process. It should be noticed that the *NOA* metric characterizes only one specific view of size, namely length, it takes no account of functionality or complexity. Also, bad process design may cause an excessive number of activities. Compared to the original `LOC` metric, the *NOA* is not language-dependent and it is easier for users to understand.

M1: NOA = Number of activities in a process

Another adaptation of the `LOC` metric is to view not only activities as program statements, but to also take into account process control-flow elements (i.e. control structures). Control-flow elements affect the execution sequence of activities. This statements are different since they are executed for their effect and do not have values. Two types of metrics can be designed depending on the structured of process.

On the one hand, we can consider that processes are well-structured [1]. When processes are well-structured we can simply count the control structures corresponding to splits, since it is explicitly known that a corresponding join exits. Please note that the structure of well-structured processes is analogue to software programs. In computer programming, a statement block is a section of code which is grouped together, much like a paragraph; such blocks consist of one or more statements. For example, in a C statement blocks are enclosed by braces { and }. In Pascal, blocks are denoted by `begin` and `end` statements. Having these characteristics in mind we design our second metric (*M2*) which counts the activities and control-flow elements of a process:

M2: NOAC = Number of activities and control-flow elements in a process

On the other hand, we also have to consider that some languages allow the construction of processes that are not well-structured. As we have already mentioned, examples of such languages include EPC and Workflow nets. In these modeling languages, splits do not have to match a corresponding join. These processes are generally more difficult to understand and result often in design errors. For processes that are not well-structured we can design a third metric (*M3*) which counts the number of activities and the number of splits and joins of a process.

M3: NOAJS = Number of activities, joins, and splits in a process

In EPC models, we would count the number of activities, XOR-joins and -splits, OR-joins and -splits, and AND-joins and -splits to calculate *NOAJS*.

2.5 Adapting McCabe's Cyclomatic Complexity

An early measure, proposed by McCabe [15], views program complexity related to the number of control paths through a program module. McCabe derived a software complexity measure from graph theory using the definition of the cyclomatic number which corresponds to the number of linearly independent paths in a program. It is intended to be independent of language and language format [17]. This measure provides a single number that can be compared to the complexity of other programs.

Since its development, McCabe's cyclomatic complexity (*MCC*) has been one of the most widely accepted software metrics and has been applied to tens of millions of lines of code in both the Department of Defense (DoD) and commercial applications. The resulting base of empirical knowledge has allowed software developers to calibrate measurements of their own software and arrive at some understanding of its complexity. McCabe's cyclomatic complexity is an indication of a program module's control-flow complexity and has been found to be a reliable indicator of complexity in large software projects [25]. Considering the number of control paths through the program, a 10-line program with 10 assignment statements is easier to understand than a 10-line program with 10 if-then statements.

MCC is defined for each module to be $e - n + 2$, where e and n are the number of edges and nodes in the control flow graph, respectively. Control flow graphs describe the logic structure of software modules. The nodes represent computational statements or expressions, and the edges represent transfer of control between nodes. Each possible execution path of a software module has a corresponding path from the entry to the exit node of the module's control flow graph. For example, in Figure 1, the *MCC* of the control flow graph for the Java code described is $14 - 11 + 2 = 5$.

2.6 The CFC Metric

In our previous work [6,7] we have designed a process complexity metric that borrows some ideas from McCabe's cyclomatic complexity. Our objective was to

Node	Statement
(1)	while(x<100){
(2)	if (a[x] % 2 == 0) {
(3)	parity = 0;
	}
	else {
(4)	parity = 1;
(5)	}
(6)	switch(parity){
	case 0:
(7)	println("a[" + i + "] is even");
	case 1:
(8)	println("a[" + i + "] is odd");
	default:
(9)	println("Unexpected error");
	}
(10)	x++;
	}
(11)	p = true;

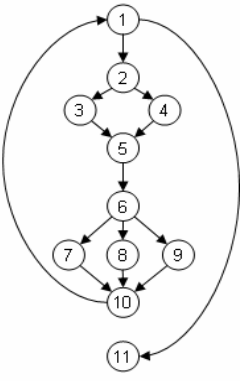

Fig. 1. of a Java program and its corresponding flowgraph

develop a metric that could be used in the same way as the MCC metric but to evaluate processes' complexity.

One of the first important observations that can be made from the MCC control flow graph, shown in Figure 1, is that this graph is extremely similar to a process. One major difference is that the nodes of a MCC control flow graph have identical semantics, while process nodes (i.e., activities) can have different semantics (e.g., AND-splits, XOR-splits, OR-joins, etc). Our approach has tackled this major difference.

The metric that we have previously developed and tested, called Control-flow Complexity (CFC) metric, was based on the analysis of XOR-splits, OR-splits, and AND-splits control-flow elements. The main idea behind the metric was to evaluate the number of mental states that have to be considered when a designer is developing a process. Splits introduce the notion of mental states in processes. When a split (XOR, OR, or AND) is introduced in a process, the business process designer has to mentally create a map or structure that accounts for the number of states that can be reached from the split. The notion of mental state is important since there are theories [19] suggesting that complexity beyond a certain point defeats the human mind's ability to perform accurate symbolic manipulations, and hence results in error.

Mathematically, the control-flow complexity metric is additive, thus it is very easy to calculate the complexity of a process, by simply adding the CFC of all split constructs. The control-flow complexity was calculated as follows, where P is a process and a an activity.

$$CFC(P) = \sum_{a \in P, a \text{ isa } xor-split} CFC_{XOR}(a) + \sum_{a \in P, a \text{ isa } or-split} CFC_{OR}(a) + \sum_{a \in P, a \text{ isa } and-split} CFC_{AND}(a)$$

The $CFC_{XOR} - split$, $CFC_{OR} - split$, and $CFC_{AND} - split$ functions is calculated as follows:

- $CFC_{XOR} - split(a) = fan - out(a)$. The control-flow complexity of XOR-splits is determined by the number of branches that can be taken.
- $CFC_{OR} - split(a) = 2^{fan-out(a)} - 1$. The control-flow complexity of OR-splits is determined by the number of states that may arise from the execution of an OR-split construct.
- $CFC_{AND} - split(a) = 1$. For an AND-split, the complexity is simply 1.

The higher the value of $CFC_{XOR} - split$, $CFC_{OR} - split$, and $CFC_{AND} - split$, the more complex is a process design, since developer has to handle all the states between control-flow constructs (splits) and their associated outgoing transitions and activities. Each formula to calculate the complexity of a split construct is based on the number of states that follow the construct. CFC analysis seeks to evaluate complexity without direct execution of processes.

The advantages of the CFC metric is that it can be used as a maintenance and quality metric, it gives the relative complexity of process designs, and it is easy to apply. Disadvantages of the CFC metric include the inability to measure data complexity, only control-flow complexity is measured. Additionally, the same weight is placed on nested and non-nested loops. However, deeply nested conditional structures are harder to understand than non-nested structures.

2.7 Adapting the Halstead Complexity Metric

The measures of Halstead [10] are the best known and most thoroughly studied composite measure of software complexity. The measures were developed as a means of determining a quantitative measure of complexity based on a program comprehension as a function of program operands (variables and constants) and operators (arithmetic operators and keywords which alter program control-flow). Halstead's metrics comprise a set of primitive measures (n1, n2, N1, and N2) that may be derived from the source code:

- n1 = number of unique operators (if, while, =, ECHO, etc);
- n2 = number of unique operands (variables or constants);
- N1 = total number of operator occurrences;
- N2 = total number of operand occurrences.

In our work, we suggest to map business process elements to the set of primitive measures proposed by Halstead. For example, n1 is the number of unique activities, splits and joins, and control-flow elements (such as sequence, switch, while, etc. in BPEL) of a business process. While the variable n2 is the number of unique data variables that are manipulated by the process and its activities. N1 and N2 can be easily derived directly from n1 and n2. With these primitive measures we introduce the notion of Halstead-based Process Complexity (*HPC*) measures for estimating process length, volume, and difficulty. These measures are based on Halstead measures and are calculates as follows:

- Process Length: N = n1*log2(n1) + n2*log2(n2)
- Process Volume: V = (N1+N2)*log2(n1+n2)
- Process Difficulty: D = (n1/2)*(N2/n2)

By the means of the presented mapping we can design an additional measure for processes based on the original measurement proposed by Halstead, including the process level, effort to implement, time to implement, and number of delivered bugs. We do not formalize these measurements since they require calibration that can only be done with empirical experiments.

Using *HPC* measures for processes has several advantages. The measures do not require in-depth analysis of process structures, they can predict rate of errors and maintenance effort, they are simple to calculate, and they can be used for most process modelling languages.

2.8 Adapting the Information Flow Metric by Henry and Kafura

Henry and Kafura [12] proposed a metric based on the impact of the information flow in a program' structure. The technique suggests identifying the number of calls to a module (i.e. the flows of local information entering: fan-in) and identifying the number of calls from a module (i.e. the flows of local information leaving: fan-out). The complexity of a procedure (PC) is defined as:

$$PC = \text{Length} * (\text{Fan-in} * \text{Fan-out})^2$$

The value of the variable length can be obtained by applying the lines of code or alternatively the McCabe's cyclomatic complexity metric. The procedure complexities are used to establish module complexities. A module with respect to a data structure DS consists of those procedures which either directly update DS or directly retrieve information from DS. As it can be seen, the measure is sensitive to the decomposition of the program into procedures and functions, on the size and the flow of information into procedures and out of procedures.

Henry and Kafura metric can be adapted to evaluate the complexity of processes in the following way. To calculate the length of an activity we need first to identify if activities are seen as black boxes or white boxes by the business process management system. If activities are black boxes then only their interface is known. Therefore, it is not possible to calculate the length of an activity. In this situation we assume the length to be 1. If activities are white boxes then the length of an activity is based on knowledge of its source code. In this situation, the length can be calculated using traditional software engineering metrics that have been previously presented, namely the `LOC` and *MCC*.

The fan-in and fan-out can be mapped directly to the inputs and outputs of activities. Activities are invoked when their inputs (fan-in) are available and the activities are scheduled for execution. When an activity completes its execution, its output data is transferred to the activities connected to it through transitions. We propose a metric called interface complexity (IC) of an activity which is defined as:

IC = Length * (number of inputs * number of ouputs)^2

The advantages of the IC metric are that it takes into account data-driven processes and it can be calculated prior to coding, during the design stage. The drawbacks of the metric are that it can give complexity values of zero if an activity has no external interactions. This typically only happens with the end activities of a process. This means the, for example, EPC processes with a large percentage of end activities will have a low complexity.

3 Cognitive Science on Software Complexity

Most approaches in the software engineering domain take certain characteristics of software as a starting point and attempt to define what effect they might have on the difficulty of the various programmer tasks (e.g. maintaining, testing and understanding code). In [5], it is argued that it is much more useful to analyse the processes involved in programmer tasks first, as well as the parameters which govern those efforts: ".. one should start with the symptoms of complexity, which are all manifested in the mind, and attempt to understand the processes which produce such symptoms". Using results from cognitive sciences, e.g. the division of the mind into short-term and long-term memory, and the mental processes involved with programming known as "chunking" and "tracing", Cant et al. come up with a set of tentative complexity metrics for software programs [5].

A similar approach for determining the complexity of a process model would be to determine meaningful process model "chunks", which can be captured as a single section in the short-term memory. One could think of constructions like a (short) sequence of activities or a control construct like an XOR-split. Each of these "chunks" would have to be characterized by a complexity score. The work in [5] suggests that notably the size of the chunk would be a good estimate. Next, it is necessary to see the control flow through these chunks, as people need to scan the relations between chunks to understand the complete picture. This is referred to as the "tracing" mechanism. In [5], not only the length of the path but also the kind of dependency influences the comprehension of the flow between chunks. For software, for example, Cant et al. state that "a conditional control structure is more complex to understand than a normal sequential section of code". For a process model, this could mean that both (a) the distance between the chunks and (b) a complexity factor for the specific kind of dependency should be used. Unfortunately, the work in [5] rather sets an agenda for complexity metrics than providing exact measures. Therefore, it is far from straightforward to transfer the presented, tentative relations to the process modelling domain.

4 Complexity of the Process Graph

Graph theory provides a rich set of graph metrics or graph measures that can be adapted for calculation of the complexity of the process graph. In [14] the

coefficient of network complexity (CNC), the complexity index (CI), the restrictiveness estimator (RT), and the number of trees in a graph are discussed as suitable for business process models.

The coefficient of network complexity (CNC) provides a rather simple metric for the complexity of a graph. It can easily be calculated as the number of arcs divided by the number of nodes. In the context of a business process model, the number of arcs has to be divided by the number of activities, joins, and splits. In formal esthetics this coefficient is also considered with the notion of elegance [21].

CNC = number of arcs / (number of activities, joins, and splits)

The complexity index (CI), or reduction complexity is defined as the minimal number of node reductions that reduces the graph to a single node. This measure shares so similarity to the notion of structuredness of a process graph and respective reduction rules. In a BPEL process it can be associated with the number of structured activities. The complexity index of a process graph has to be calculated algorithmically and is not applicable for process models with arbitrary cycles.

Restrictiveness estimator (RT) is an estimator for the number of feasible sequences in a graph. RT requires the reachability matrix r_{ij}, i.e. the transitive closure of the adjacency matrix, to be calculated.

$$\mathrm{RT} = 2\Sigma r_{ij} - 6(N-1)/(N-2)(N-3)$$

There are further measures in graph theory which demand rather complex computations. The number of trees in a graph requires the tree-generating determinant to be calculated based on the adjacency matrix (see [14]). Measures such as tree width, directed tree width, and directed acyclic graph width are compared in [22]. The latter measures how close a graph is to a directed acyclic graph.

5 Contributions and Limitations

In this paper, we have surveyed several contributions from software engineering, cognitive science, and graph theory, and we discussed to what extent analogous metrics and measurements can be defined for business process models. In order to demonstrate that these metrics serves their purpose, we plan to carry out several empirical validations by means of controlled experiments. These experiments will involve more than 100 students from the Eindhoven University of Technology (Netherlands), the Vienna University of Economics and Business Administration (Austria), and the University of Madeira (Portugal). The collected data will be analyzed using statistical methods to verify the degree of correlation between students' perception of the complexity of processes and the proposed metrics. It should be noted that we have already conducted a small

experiment that involved 19 graduate students in Computer Science, as part of a research project, and tested if the control-flow complexity of a set of 22 business processes could be predicted using the CFC metric. Analyzing the collected data using statistical methods we have concluded that the CFC metric is highly correlated with the control-flow complexity of processes. This metric can, therefore, be used by business process analysts and process designers to analyze the complexity of processes and, if possible, develop simpler processes.

References

1. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.
2. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems, 2003.
3. M. Azuma and D. Mole. Software management practice and metrics in the european community and japan: Some results of a survey. *Journal of Systems and Software*, 26(1):5–18, 1994.
4. F. B. Bastani. An approach to measuring program complexity. *COMPSAC '83*, pages 1–8, 1983.
5. S. N. Cant, D. R. Jeffery, and B. Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7).
6. J. Cardoso. Control-flow Complexity Measurement of Processes and Weyuker's Properties. In *6th International Enformatika Conference*, Transactions on Enformatika, Systems Sciences and Engineering, Vol. 8, pages 213–218, 2005.
7. J. Cardoso. *Workflow Handbook 2005*, chapter Evaluating Workflows and Web Process Complexity, pages 284–290. Future Strategies, Inc., Lighthouse Point, FL, USA, 2005.
8. J. Cardoso. Complexity analysis of bpel web processes. *Journal of Software Process: Improvement and Practice*, 2006. to appear.
9. A.S. Gucegliooglu and O.W. Demiros. Using Software Quality Characteristics to Measure Business Process Quality. In W.M.P. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Business Process Management (BPM 2005)*, volume 3649, pages 374–379. Springer-Verlag, Berlin, 2005.
10. M. H. Halstead. *Elements of Software Science*. Elsevier, Amsterdam, 1987.
11. W. Harrison and K. Magel. A topological analysis of computer programs with less than three binary branches. *ACM SIGPLAN Notices*, april:51–63, 1981.
12. S. Henry and D. Kafura. Software structure metrics based on information-flow. *IEEE Transactions On Software Engineering*, 7(5):510–518, 1981.
13. G. E. Kalb. Counting lines of code, confusions, conclusions, and recommendations. Briefing to the 3rd Annual REVIC User's Group Conference, 1990.
14. Antti M. Latva-Koivisto. Finding a complexity for business process models. Research report, Helsinki University of Technology, February 2001.
15. T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.
16. T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32:1415–1425, 1989.

17. T. J. McCabe and A. H. Watson. Software complexity. *Journal of Defence Software Engineering*, 7(12):5–9, 1994. Crosstalk.
18. J. Mendling, M. Moser, G. Neumann, H.M.W. Verbeek, and B.F. van Dongen W.M.P. van der Aalst. A Quantitative Analysis of Faulty EPCs in the SAP Reference Model. BPM Center Report BPM-06-08, Eindhoven University of Technology, Eindhoven, 2006.
19. G. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 1956.
20. Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, 2006.
21. G. Neumann. *Metaprogrammierung und Prolog*. Addison-Wesley, December 1988.
22. Jan Obdrzalek. Dag-width: connectivity measure for directed graphs. In *Symposium on Discrete Algorithms*, pages 814–821. ACM Press, 2006.
23. H.A. Reijers and Irene T.P. Vanderfeesten. Cohesion and Coupling Metrics for Workflow Process Design. In J. Desel, B. Pernici, and M. Weske, editors, *Business Process Management (BPM 2004)*, volume 3080, pages 290–305. Springer-Verlag, Berlin, 2004.
24. M. Shepperd. Early life-cycle metrics and software quality models. *Information and Software Technology*, 32(4):311–316, 1990.
25. W. Ward. Software defect prevention using mccabe's complexity metric. *Hewlett Packard Journal*, 40(2):64–69, 1989.
26. H. Zuse. *Software Complexity: Measures and Methods*. Walter de Gruyter and Co, New Jersey, 1991.