Complexity Metrics for Business Process Models

Volker Gruhn and Ralf Laue

Chair of Applied Telematics / e-Business*
Computer Science Faculty, University of Leipzig, Germany
{gruhn,laue}@ebus.informatik.uni-leipzig.de

Abstract. Business process models, often modelled using graphical languages like UML, serve as a base for communication between the stakeholders in the software development process. To fulfill this purpose, they should be easy to understand and easy to maintain.
For this reason, it is useful to have measures that can give us some information about understandability and maintainability of a business process model. Such measures should tell us whether the model has an appropriate size, is clearly structured, easy to comprehend and partitioned into modules in a sensible way.
This paper discusses how existing research results on the complexity of software can be extended in order to analyze the complexity of business process models.

1 Introduction

One of the main purposes for developing business process models (BPM) is to support the communication between the stakeholders in the software development process (domain experts, business process analysts, software developers to name just a few). To fulfill this purpose, the models should be easy to understand and easy to maintain. If we want to create models that are easy to understand, at first we have to define what "easy to understand" means: We are interested in complexity metrics, i.e. measurements that can tell us whether a model is easy or difficult to understand. In the latter case, we may conclude from the metrics that the model should be re-engineered, for example by decomposing it into simpler modules.

A significant amount of research has been done on the complexity of software programs, and software complexity metrics have been used successfully for purposes like predicting the error rate, estimating maintenance costs or identifying pieces of software that should be re-engineered. In this paper, we discuss how the ideas known from software complexity research can be used for analyzing the complexity of BPMs. To our best knowledge, there is almost no published work about this subject: Other papers like [1] discussed rather simple BPM languages (like process charts) only. Cardoso[2] (whose approach we discuss in section 3) seems to be the only author so far who addresses the problems of measuring

* The Chair of Applied Telematics / e-Business is endowed by Deutsche Telekom AG

the complexity of more expressive BPM languages. However, we expect that the results of the research by Baroni[3] will be useful for BPM as well.

We want to give an overview about factors that have an influence on the complexity of a BPM and metrics that can be used to measure these factors. Validation of the proposed metrics in a case study is beyond of the scope of this paper and has to be be done in future research. Also we restrict our discussion to the control flow of the BPM, neglecting other aspects like data flow and resource utilization.

Even if the measurement of the complexity of *business process* models is the main purpose of this paper, it is worth mentioning that the main ideas should work in the same way for other kinds of models which can express a control flows with parallel activities and decisions.

The metrics discussed in this paper are independent from the modeling language. They can be used for models in various formalisms, for example event driven process chains[4], UML activity diagrams[5], BPMN[6] or YAWL[7]. Of course, these graphical languages have different expressiveness. This paper discusses only those elements which can be found in all languages. Additional research would be necessary to address advanced concepts that are supported by some but not all business process modeling languages. In particular, the use of concepts like Exceptions, Cancellation or Compensation can increase the difficulty of a BPM considerably.

If we have to depict a BPM in this article, we use the notation of event driven process chains[4], mainly because of its simplicity. EPCs consist of functions (activities which need to be executed, depicted as rounded boxes), events (pre- and postconditions before / after a function is executed, depicted as hexagons) and connectors (which can split or join the flow of control between the elements). Arcs between these elements represent the control flow. The connectors are used to model parallel and alternative executions. There are two kinds of connectors: Splits have one incoming and at least two outgoing arcs, joins have at least two incoming arcs and one outgoing arc.

AND-connectors (depicted as ) are used to model parallel execution. When an AND-split is executed, the elements on all outgoing arcs have to be executed in parallel. The corresponding AND-join connector waits until all parallel control flows that have been started are finished.

XOR-connectors (depicted as ) can be used to model alternative execution: A XOR-split has multiple outgoing arcs, but only one of them will be processed. The corresponding XOR-join waits for the completition of the control flow on the selected arc.

Finally, OR-connectors (depicted as ) are used to model parallel execution of one or more flows. An OR-split starts the processing of one or more of its outgoing arcs. The corresponding OR-join waits until all control flows that have been started by the coresponding OR-split are finished.

In the following sections, we discuss well-known complexity metrics for software programs and their adaption to BPMs. Section 2 discusses the *Lines of Code* as the simplest complexity metric. Metrics, which take into account the

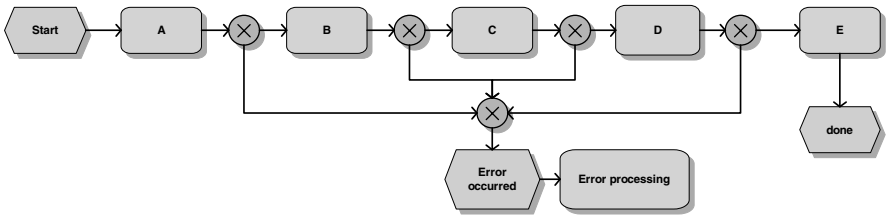

Fig. 1. "almost linear" model with CFC=8

control flow and the structure of a model, are discussed in sections 3 and 4. Section 5 presents metrics which measure the cognitive effort to comprehend a model. Metrics for a BPMs that are decomposed into modules are discussed in chapter 6. Finally, chapter 7 gives a summary about the metrics and their usage for analyzing the complexity of BPMs. The related work on software metrics is discussed throughout this paper, for this reason there is no extra "related work" section.

2 Size of the Model: Lines of Code

The IEEE Standard Computer Dictionary defines complexity as "the degree to which a system or component has a design or implementation that is difficult to understand and verify"[8]. The easiest complexity measurement for software is the "lines of code" (LOC) count which represents the program size. While for assembler programs a line of code is the same as an instruction statement, for programs written in a modern programming language, the LOC count usually refers to the number of executable statements (ignoring comments, line breaks etc.)[9].

For BPMs, the number of activities in the model can be regarded as an equivalent to the number of executable statements in a piece of software. For this reason, the "number of activities" is a simple, easy to understand measure for the size of a BPM.

However, the "number of activities" metric does not take into account the structure of the model: A BPM with 50 activities may be written using a well-structured control flow which is easy to understand or in an unstructured way which makes understanding very hard. For this reason, we will discuss other metrics which measure the complexity of the control flow in the next chapters.

3 Control Flow Complexity of the Model: McCabe-Metric

The cyclomatic number, introduced by McCabe[10], is one of the most widely used software metrics. It is calculated from the control flow graph and measures

the number of linearly-independent paths through a program. For a formal definition, we refer to the literature[9, 10]; informally it is sufficient to say that the cyclomatic number is equal to the number of binary decisions (for example `IF`-statements in a programming language) plus 1. Non-binary decisions (for example `select` or `case`-statements in a programming language) with n possible results are counted as n-1 binary decisions.

The cyclomatic number measures the number of all possible control flows in a program. For this reason, a low cyclomatic number indicates that the program is easy to understand and to modify. The cyclomatic number is also an indicator of testability, because it corresponds to the number of test cases needed to achieve full path coverage. [11] found that there is a significant correlation between the cyclomatic number of a piece of software and its defect level.

Cardoso[2] (to our knowledge the only other author who has published about complexity analysis of workflows / BPM) has suggested a complexity measure for BPMs which is a generalization of McCabe's cyclomatic number. The control-flow complexity of processes (CFC) as defined by Cardoso is the number of mental states that have to be considered when a designer develops a process. Analogously to the cyclomatic number which is equal to the number of binary decisions plus 1, the corresponding CFC metric for BPMs counts the number of decisions in the flow of control. Every split in the modell adds to the number of possible decisions as follows:

- AND-split: As always *all* transitions outgoing from an AND-split must be processed, the designer needs only to consider one state as the result of the execution of an AND-split. For this reason, every AND-split in a model adds 1 to the CFC metric of this model.
- XOR-split with n outgoing transitions: Exactly one from n possible paths must be taken, i.e. the designer has to consider n possible states that may arise from the execution of the XOR-split. For this reason, every XOR-split with n outgoing transitions adds n to the CFC metric of this model.
- OR-split with n outgoing transitions: There are $2^n - 1$ possibilities to process at least one and at most n of the outgoing transitions of an OR-split, i.e. every OR-split with n outgoing transitions adds $2^n - 1$ to the CFC metric.

[2] reports the promising result of a first (yet rather small) experiment to verify the validity of the CFC metric: A correlation was found between the "perceived complexity" as rated by students and the CFC metric.

A shortcoming of this metric is that just counting the number of possible decisions in a model gives only few information about its structure. For example, compare the BPMs in Fig. 1 and Fig. 2. For both models, the CFC is 8, because they contain the same number of binary decisions. Nevertheless, the "almost linear" model in 1 is obviously much easier to understand.

As another example, Fig. 3 shows two models with one OR split/join-block. The left one has a CFC of 3, the right one a CFC of 15. If we want to measure the difficulty to comprehend a model, this difference may be unjustified.

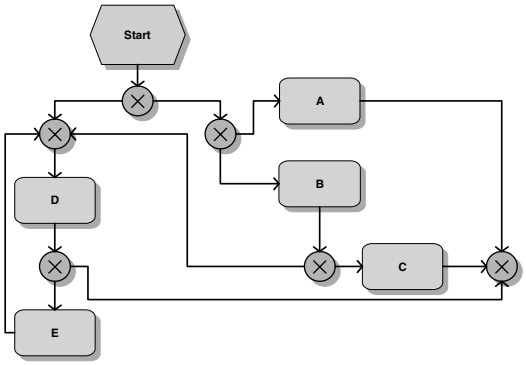

Fig. 2. "unstructured" model with CFC=8

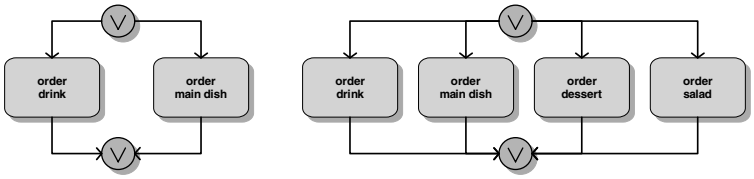

Fig. 3. Two models which different CFC metrics

To overcome these shortcomings, we will discuss other metrics in the next chapters. These metrics take into account the structure of the BPM. They can be used additionally to the CFC metrics.

4 Structure of the Model: Nesting Depth and Jumps out of a Control Structure

Both models depicted in Fig. 1 and Fig. 2 have a CFC metric of 8. However, there should be no doubt that Fig. 1 shows a much less complex model than Fig. 2. One reason for this lies in the fact that that Fig. 1 shows an "almost linear" flow of control while in Fig. 2, there are several nested XOR-splits and XOR-joins. From the research about software complexity, we know that the metrics "maximum nesting depth" and "mean nesting depth" are suitable for measuring this factor which has influence on the overall complexity of the model: A greater nesting depth implies greater complexity. [12] showed that both nesting depth metrics have a strong influence on other structure-related complexity metrics.

The definition of the metrics "maximum nesting depth" and "mean nesting depth" for BPMs is straightforward: The nesting depth of an action is the number of decisions in the control flow that are necessary to perform this action. The

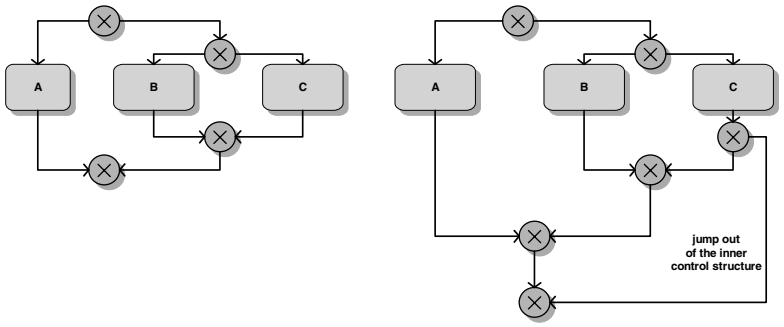

Fig. 4. The left model has properly nested control flow blocks, the right one has not.

maximum nesting depth in Fig. 1 is 1 (every error leads to the error handling process), the model in Fig. 2 has a maximum nesting depth of 3 (because three decisions must be made in order to process action C). Using this metric, we conclude that Fig. 2 shows a more complex model than Fig. 1. The nesting depth metric can be calculated additionally to the CFC metric that has been discussed in the last chapter.

We have to note that the use of the term "nesting depth" is a little bit misleading. Other than modern structured programming languages, common graph-oriented business modeling languages (for example UML activity diagrams[5] or YAWL[7]) do not require proper nesting, i.e. splits and joins does not have to occur pairwise. This is comparable with programming languages that do not only allow structured loops (like `repeat...until` etc.) but also arbitrary `GOTO`-jumps. Not without a reason, [13] writes that "the current unstructured style of business process modeling, which we can call spaghetti business process modeling, leads to similar problems as spaghetti coding".

If we are asking for the complexity of a BPM, we must take into account these difficulties that arise from splits and joins that do not occur pairwise in well-nested constructs.

As an example, Fig. 4 shows two similar models. The splits and joins in the left one are properly nested - the inner control structure (XOR-split/XOR-join) is contained completely within the outer control structure. In the right model, there is a jump out of the inner control block, and this jumps leads to a target "behind" the outer control structure.

For software programs, [14] introduced the *knot count* metric for measuring such (undesirable) jumps out of and into a structured control flow. A control graph of a program has a knot whenever the paths associated with transfer of control intersect. For example, this is the case for the control flow graph shown in Fig. 5.

For BPMs, van der Aalst[15] defined the term well-structuredness: A model is well-structured if the split/join constructions are properly nested. Formally,

Fig. 5. Control flow with a knot

well-structuredness is defined in terms of Petri net terminology. [15] models a BPM as workflow net (a special case of a Petri net). It is defined to be well-structured if this workflow net does not contain handles.[1] This means that the number of handles in the workflow net is a measure for the number of not well-structured constructs in this model. It is comparable with the knot count metric that has been used successfully for measuring software complexity. In order to compute this metric for a BPM, we have to transform this model into a workflow net. This can be done for example for BPMs modelled as event-driven process chains[17].

In general, using not well-structured models as the one in Fig. 4 can be regarded as bad modeling style which makes understanding of the model more complicate. Mostly, it is possible to redesign such models by using well-structured ones[18]. For example, the language BPEL4WS[19] or the workflow management system ADEPT[20] have semantic restrictions which force the modeler to build well-structured models.[2] For such models, the metric "number of handles" is always 0, just as the knot count metric is always 0 for programs written in a structured programming language

5 Comprehensiveness of the model: Cognitive Complexity Metrics

In chapter 3, we have used two examples to illustrate shortcomings of the CFC[2] metric. The additional metrics proposed in chapter 4 are helpful in the example depicted in Fig. 1 and Fig. 2. They could be used in addition to the CFC metric.

Now let's have a look at the other example (Fig. 3). Both models show a control structure in which one or more paths are executed in parallel. If we define complexity as "difficulty to test" (i.e. number of test cases needed to achieve full path coverage), the CFC metric does a perfect job. However, we argue that this metric is less useful if we define complexity as "difficulty to *understand* a model": The number of control flow paths between the OR-split and the OR-join has not

[1] A Petri net has a handle, iff for any pair of nodes x and y such that one of the nodes is a place and the other a transition, there exist two different paths from x to y which have more common elements than just x and y. Details can be found in [16] and [15]. We define the number of handles in a workflow net as the number of pairs of nodes with the mentioned property.

[2] It is worth mentioning that the UML 2.0 specification does not require anymore that forks and joins must occur pairwise, which was necessary in the version 1.0 of the standard. We do not think that this can be seen as a progress towards better understandable models with less errors.

too much influence on the effort that is necessary to comprehend this control structure. Regardless of whether there are 2, 4 or 10 control flow paths between the split and the join, the person who reads the model will always understand everything between the OR-split and the OR-join as only *one control structure*.

Shao and Wang [21] defined the cognitive weight as a metric to measure the effort required for comprehending a piece of software. Based on empirical studies, they defined *cognitive weights* for basic control structures. Table 1 shows the basic control structures and its cognitive weights which are a measure for the difficulty to understand a control structure.

The cognitive weight of a basic control structure is a measure for the difficulty to understand this control structure.

control structures	W_i
sequence (an arbitrary number of statements in a sequence without branching)	1
call of an user-defined function	2
branching with if-then or if-then-else	2
branching with case (with an arbitrary number of selectable cases)	3
Iteration (for-do, repeat-until, while-do)	3
recursive function call	3
execution of control flows in parallel	4
Interrupt	4

Table 1. cognitive weights as defined in [21]

The cognitive weight of a software component without nested control structures is defined as the sum of the cognitive weights of its control structures according to table 1. It seems to be a promising approach to use the ideas from [21] to define cognitive weights for BPMs. If we want to do so, we have to consider the fact that BPMs may be modelled in an unstructured way as discussed in chapter 4. Also, table 1 should be tailored to the needs of BPMs: While recursion does not play any role in business process modeling, it is necessary to consider other concepts like cancellation or the multi-choice-pattern[22]. This will be subject of our further research, which should include validating experiments as well.

The idea behind cognitive weights is to regard basic control structures as *patterns* that can be understood by the reader as a whole. A similar idea has been proposed by [23]. This approach is based on automatically finding well-known architectural patterns (for example from [24]) in a UML model. The idea behind this approach is that well-documented patterns have been found highly mature and using them helps to improve code quality, understandability and maintainability. Obviously, this assertion should be regarded with care: Architectural patterns are only useful if they are used by experienced programmers in the right way, and an extensive use of patterns does not have to mean anything for the quality of the code. So, if the approach from [23] is used, a deep knowledge about the patterns and their correct usage is necessary.

However, [23] does not only discuss the use of "good" design patterns, it also recognizes so called anti-patterns, i.e. commonly occurring solutions to a problem that are known to have negative consequences. If such an anti-pattern is found in the code, this can be regarded as a sign of bad programming.

It seems to be appealing to us to use the ideas from [23] in the context of business process models. In particular, finding anti-patterns should be very useful in order to uncover bad modeling style. An example for such an anti-pattern for BPMs is the *Implicit Termination* pattern as described in [22]. (Implicit Termination means that a process should be terminated if nothing else is to do - without modeling an explicit end state, see [22] for details).

6 Modularization of the Model: Fan-in / Fan-out-Metrics

Modular modeling of business processes is supported by all major BPM languages (for example by nested activity diagrams in UML, sub-processes in BPMN, composite tasks in YAWL or hierarchical event-driven process chains).

Dividing a BPM in modular sub-models cannot only help to make the BPM easier to understand, it can also lead to smaller, reusable models – if modularization is used in a reasonable way.

For analyzing the modularization of a BPM, we can adopt the ideas of Henry and Kafura[25] who developed metrics for reasoning about the structure of modularized software systems. They measure the fan-in and fan out for every module, where *fan-in* is a count of all other modules that call a given module and *fan-out* is a count of all other modules that are called from the model under investigation. Usually, the modules with a large fan-in are small submodules doing some simple job that is needed by a lot of other modules. On the other hand, the modules with a large fan-out are large modules on the higher layers of the design structure. If a module with both large fan-in and fan-out is detected, this may indicate that a re-design could improve the model. [25] suggests the metric $((fan-in)\cdot(fan-out))^2$ in order to measure this kind of structural complexity.

This metric can be used in the same way for analyzing BPMs: If a submodel of a BPM has a high structural complexity according to the fan-in/fan-out metric, they will be difficult to use and are most likely poorly designed.

7 Conclusion and Directions for Future Research

Table 2 summarizes the results from the past chapters: Metrics for measuring the complexity of *software* are compared with corresponding metrics for *business process models*. Also, we assess the significance of these metrics for BPM. As the metrics in the left column of the table have been proved to be useful for measuring the complexity of software, we expect that the corresponding metrics in the second column will be useful for measuring the complexity of BPMs. It is one aim of our future research to validate this expectation.

software complexity metric	**corresponding metric for BPM**	**usage, significance**
Lines of Code	number of activities	very simple, does not take into account the control-flow
cyclomatic number [10]	CFC as defined by Cardoso [2]	measures the number of possible control flow decisions, well-suited for measuring the number of test cases needed to test the model, does not take into account other structure-related information
max. / mean nesting depth	max. / mean nesting depth	provides information about structure, can be used complementary to the CFC metric
knot-count [14]	number of handles	measure of "well-structuredness" (for example jumps out of or into control-flow structures) is always 0 for well-structured models can be used complementary to the CFC metric
cognitive weight [21]	cognitive weight (tailored for BPM)	measures the cognitive effort to understand a model, can indicate that a model should be re-designed
(Anti)Patterns [26]	(Anti)Patterns for BPM	experience with the patterns needed counting the usage of anti-patterns in a BPM can help to detect poor modeling
Fan-in / Fan-out [25]	Fan-in / Fan-out	can indicate poor modularization

Table 2. complexity metrics for software and BPM

Due to the number of factors that contribute to the complexity of a BPM, we cannot identify a single metric that measures all aspects of a model's complexity. This situation is well-known from the measuring of software complexity. A common solution is to associate different metrics within a metrics suite. Each individual metric in the suite measures one aspect of the complexity, and together they give a more accurate overview. For BPMs, the CFC as defined by Cardoso[2] or a cognitive weight metric (tailored to BPM) seems to be suitable to give some kind of "general" information about the complexity of a BPM. Nesting depth can be used complementary to both CFC and cognitive weight. Additionally, the number of handles and anti-patterns are useful to uncover bad modeling style, modeling errors and models that are difficult to understand. The fan-in/fan-out metrics can be used to evaluate the decomposition of the model into sub-models.

All metrics discussed in this paper can be easily computed by a machine. Also, they are independent from the modeling language, because they use "high-level" information from the control flow graph of the BPM only.

The layout of a graphical model and the comprehensiveness of the texts used in the model are aspects of complexity that cannot be measured with the metrics discussed in this paper. Both play a very important role anyway. Important considerations to good layout include for example:

- choosing size and color of the graphical elements in the model with care
- modeling time-dependency horizontally from left to right (at least for people who speak languages that read left-to-right) or vertically from top to bottom
- aligning the edges of the graphical elements
- avoiding intersecting arrows

All these points are important for drawing easy-to-read graphical BPMs. However, they are beyond the scope of this paper. The same holds for the quality of texts that are used in the model, for example to describe the activities.

Most metrics discussed in this paper can instantly be used for analyzing BPM, namely the number of activities, Cardoso's CFC[2], the max. / mean nesting depth, the number of handles and the fan-in/fan-out metrics. For defining and using cognitive weights, patterns and anti-patterns, further tailoring to the demands of BPM is necessary. This will be the subject of our ongoing research as well as the study of the relations between data-flow and complexity.

References

1. Latva-Koivisto, A.: Finding a complexity measure for business process models (2002)
2. Cardoso, J.: How to measure the control-flow complexity of web processes and workflows. In: The Workflow Handbook. (2005) 199–212
3. Baroni, A.L.: Quantitative assessment of UML dynamic models. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2005) 366–369
4. Aalst, W.: Formalization and verification of event-driven process chains. Information & Software Technology **41**(10) (1999) 639–650
5. Object Management Group: UML 2.0 Superstructure Final Adopted Specification. Technical report (2003)
6. Business Process Management Initiative: Business Process Modeling Notation. Technical report, BPMI.org (2004)
7. Aalst, W., Hofstede, A.: YAWL: Yet another workflow language. Technical Report FIT-TR-2002-06, Queensland University of Technology, Brisbane (2002)
8. Institute of Electrical and Electronics Engineers: IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. (1990)
9. Kan, S.H.: Metrics and Models in Software Quality Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
10. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. **2**(4) (1976) 308–320
11. Grady, R.B.: Successfully applying software metrics. Computer **27**(9) (1994) 18–25
12. Schroeder, A.: Integrated program measurement and documentation tools. In: ICSE '84: Proceedings of the 7th international conference on Software engineering, Piscataway, NJ, USA, IEEE Press (1984) 304–313
13. Holl, A., Valentin, G.: Structured business process modeling (SBPM). In: Information Systems Research in Scandinavia (IRIS 27) (CD-ROM). (2004)
14. M. R. Woodward, M.A.H., Hedley, D.: A measure of control-flow complexity in program text. IEEE Transactions on Software Engineering **SE-5**(1) (1979) 45–50

15. Aalst, W.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers **8**(1) (1998) 21–66
16. Esparza, J., Silva, M.: Circuits, handles, bridges and nets. In: Applications and Theory of Petri Nets. (1989) 210–242
17. Dehnert, J., Aalst, W.: Bridging The Gap Between Business Models And Workflow Specifications. Int. J. Cooperative Inf. Syst. **13**(3) (2004) 289–332
18. Kiepuszewski, B., ter Hofstede, A.H.M., Bussler, C.: On structured workflow modelling. In: Conference on Advanced Information Systems Engineering. (2000) 431–445
19. Andrews, T.: Business process execution language for web services. (2003)
20. Reichert, M., Dadam, P.: ADEPTflex -supporting dynamic changes of workflows without losing control. Journal of Intelligent Information Systems **10**(2) (1998) 93–129
21. Shao, J., Wang, Y.: A new measure of software complexity based on cognitive weights. IEEE Canadian Journal of Electrical and Computer Engineering **28**(2) (2003) 69–74
22. Aalst, W., Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases **14**(3) (2003)
23. Gustafsson, J.: Metrics calculation in MAISA (2000)
24. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: Abstraction and reuse of object-oriented design. Lecture Notes in Computer Science **707** (1993) 406–431
25. Henry, S., Kafura, K.: Software structure metrics based on information flow. IEEE Transactions on Software Engineering **7(5)** (1981) 510–518
26. Paakki, J., Karhinen, A., Gustafsson, J., Nenonen, L., Verkamo, A.I.: Software metrics by architectural pattern mining. In: Proc. International Conference on Software: Theory and Practice. (2000) 325–332