

Declarative versus Imperative Process Modeling Languages: The Issue of Maintainability

Dirk Fahland¹, Jan Mendling¹, Hajo A. Reijers², Barbara Weber³,
Matthias Weidlich⁴, and Stefan Zugal³

¹ Humboldt-Universität zu Berlin, Germany
fahland@informatik.hu-berlin.de | jan.mendling@wiwi.hu-berlin.de

² Eindhoven University of Technology, The Netherlands
h.a.reijers@tue.nl

³ University of Innsbruck, Austria
barbara.weber@uibk.ac.at | stefan.zugal@uibk.ac.at

⁴ Hasso-Plattner-Institute, University of Potsdam, Germany
matthias.weidlich@hpi.uni-potsdam.de

Abstract. The rise of interest in declarative languages for process modeling both justifies and demands empirical investigations into their presumed advantages over more traditional, imperative alternatives. Our concern in this paper is with the ease of maintaining business process models, for example due to changing performance or conformance demands. We aim to contribute to a rigorous, theoretical discussion of this topic by drawing a link to well-established research on maintainability of information artifacts.

Keywords: Process model maintainability, declarative versus imperative modeling, cognitive dimensions framework

1 Introduction

The ongoing release of new modeling languages is an important challenge for process modeling. How should we weigh the claims in favor of new languages that relate to “ease of use”? In [1] a new process modeling language is proposed that is claimed to be “easily human-readable”. UML Activity Diagrams, EPCs, IDEF3, and YAWL, in contrast, are considered in that work to be too difficult to apply for capturing real-life processes, leading to models that would be difficult to interpret. Our intent is not to repudiate such claims, nor to dispute the need for new process modeling languages. Rather, our point is that quality issues cannot be addressed by formal research alone. We are in need of theories – either by finding or establishing them – that explain how people interact with information artifacts like process models. Furthermore, an empirical research agenda is required to put the explanatory powers of such theories to the test.

In this paper, we zoom in on the spectrum of imperative¹ versus declarative process modeling languages. This distinction is arguably one of the most promi-

¹ Computer scientists prefer the term “procedural”; the term “imperative” is popular in other communities [2]. In this paper, we will be using the terms as synonyms.

ment in the development of new modeling languages. For example, with respect to the recent development of ConDec (first published as “DecSerFlow”), a declarative process modeling language, it is claimed that “many problems described in literature (e.g., the dynamic change bug and other problems for procedural languages) can be avoided, thus making change easy” [3].

In an earlier paper [4], we focused on the *understandability* of a process model as an important point of comparison between languages. The motivation for the current paper is an insight that we encountered from cognitive research into programming languages: *design is redesign* [5]. It turns out that software programs are created iteratively, i.e., any attempted solution to any goal may be subject to later change [6]. Like software programs, business processes are subject to change too (e.g., to address the need for process optimization, organizational engineering, compliance issues and market dynamics) [7]. So, the constant need for process evolution makes *maintainability* another important quality factor, just as it is in comparing programming languages. Both *understandability* issues, which we discussed in [4], and *modifiability* of process models are important factors influencing the ease of adapting a process model along evolving needs.

Against this background, our paper aims to propose how different process modeling languages affect the *maintainability* of the models that are created with these. Its contribution is a set of theoretically grounded propositions about the differences between imperative and declarative process modeling languages with respect to modification issues. As such, this paper is an essential stepping stone to an empirical evaluation of these languages, which is planned by the authors as future research. It should be noted that the focus of this paper is on build-time modifications of process models and does not consider run-time aspects of changes (e.g., ad-hoc changes or instance propagation [8, 9]). Moreover, the paper deals with the question of to what extent a particular language embraces changes and fosters maintainability, whereas the effect that the used environment has on ease of change is not considered.

To argue and support the proposed hypotheses, this paper is structured as follows. Section 2 provides a theoretical background for our work. Section 3 characterizes the notational spectrum of process modeling languages. Section 4 derives propositions on when a process modeling language could be superior to another one, based on the cognitive dimensions framework. Finally, Section 5 describes the empirical research agenda for validating the propositions.

2 Background

To the best of our knowledge, no explicit explorations have taken place of differences in the *modifiability* of process models as linked to the language that was used to create them. Limited research has investigated the impact on *understandability* [10], and found a slightly better performance of models created with EPCs than with a Petri net variant.

In lack of a theoretical basis or earlier relevant results for the process modeling domain, we turn our attention to the field of software engineering. Various au-

thors have noted the similarities between process models and software programs [11, 12]. For example, a software program is usually partitioned into modules or functions, which expect a group of inputs and provide some output. Similar to this compositional structure, a business process model consists of activities, each of which may contain smaller steps (operations) that may update the values of data objects. Furthermore, just like the interactions between modules and functions in a software program are precisely specified using various language constructs, the order of activity execution in a process model is defined using logic operators. For software programs and business process models alike, *human agents* are concerned with properly capturing and updating their logic content. This stresses how important it is that both a software program and a process model can be easily comprehended: Making sense of such artifacts is important during the construction process and while modifying them at later stages.

In the past, heated debates have taken place about the superiority of one programming language over the other, e.g. with respect to expressiveness [13] or effectiveness [14]. During the 1970s and 1980s, alternative views were proposed on how programmers make sense of code leading to various theoretical explanations about the impact programming languages have on this process. In [4], we summarized this debate and the opposing views that were brought forward.

An important outcome of this debate, and one that has been postulated and empirically validated in [15, 16, 5], is that different tasks that involve sense-making of software code are supported differently by the *same* programming language. For example, the overall impact of a modification of a single declaration may be difficult to understand in a PASCAL program, but it might be relatively easy to develop a mental picture of the control-flow for the same program. Therefore, a programming language may provide superior support with respect to one comprehension task, while it may be outperformed by other languages with respect to a different task.

The latter view has been the basis for the “mental operations theory” [5], which in essence states that a notation that requires fewer mental operations from a person for any task is the better performing one. In other words, a “matched pair” between the notational characteristics and a task gives the best performance. This view has evolved and matured over the years towards the “cognitive dimensions framework” (CDF)[6, 17], which contains many different characteristics to distinguish notations from each other. This framework has been highly influential in language usability studies [18]. The CDF extends the main postulate of the mental operations theory towards a broad evaluation tool for a wide variety of notations, e.g. spreadsheets, style sheets, diagrams, etc. As such, and in absence of other competing, domain-specific theories, it appears the best available candidate for our purposes. As far as we know, no other work in the process modeling domains has built on the CDF, with the exception of [19].

The relevance of the CDF is particularly evident when in [20] Green and Blackwell elaborate on the relation between activity types and cognitive dimensions. Figure 1 shows that several cognitive dimensions foster modifications (e.g.,

	<i>transcription</i>	<i>incrementation</i>	<i>modification</i>	<i>exploration</i>
viscosity	acceptable	acceptable	harmful	harmful
role-expressiveness	desirable	desirable	essential	essential
hidden dependencies	acceptable	acceptable	harmful	acceptable for small tasks
premature commitment	harmful	harmful	harmful	harmful
abstraction barrier	harmful	harmful	harmful	harmful
abstraction hunger	useful	useful (?)	useful	harmful
secondary notation	useful (?)	-	v. useful	?
visibility / juxtaposability	not vital	not vital	important	important

Fig. 1. Relationships between activity types (columns) and cognitive dimensions (rows)

role expressiveness or abstraction hunger), while others hinder them (e.g., viscosity, hidden dependencies, premature commitment).

Adapting a software program to evolving needs involves both sense-making tasks (i.e., to determine which changes have to be made) and action tasks (i.e., to apply the respective changes to the program). An important result that has been established in the development of the CDF regarding the sense-making of information artifacts relates to the difference between the tasks of looking for sequential and for circumstantial information in a program. While *sequential* information explains how input conditions lead to a certain outcome, *circumstantial* information relates to the overall conditions that produced that outcome. Empirical evidence supports the hypothesis that procedural programming languages display sequential information in a readily-used form, while declarative languages display circumstantial information in a readily-used form [15, 5]. The reverse is also true: Just as procedural languages tend to obscure circumstantial information, so do declarative languages obscure sequential information.

When considering “modifiability” of a software program (or an information artifact in general) several competing factors have to be considered. Most notably is the cognitive dimension of “viscosity” which refers to the ease with which changes can be achieved with a particular system. Green [6] distinguishes between *repetitive viscosity* (i.e. referring to the “resistance to change”) and *knock-on viscosity* (i.e. to what extent once having made a change entails further actions to restore consistency). Siddiqi et al. [21] extend the work of Green [6] comparing the viscosity of procedural and declarative programming languages. Both the results of Green and Siddiqi et al. point at a relatively low repetitive viscosity for BASIC compared to PROLOG in the domain of programming languages. In turn, the results for knock-on viscosity are inverse (i.e., PROLOG has a lower knock-on viscosity compared to BASIC). These results point to the fact that “resistance to change” involves a variety of competing factors. The role of tool support that is available to someone manipulating an information artifact must not be underestimated.

While viscosity is affected by both the *entity* being manipulated (notation) and the *tools* that enable the manipulation (environment) [21], other dimensions

in the CDF, like “premature commitment”, rather relate to the environment that is imposing restrictions on the ordering in which things can be inserted. Green [6] states that “No problems can arise in an environment where the statements may be inserted in any convenient order - paper and pencil, for instance.”. This is highly relevant for any experimental design to test our hypotheses.

The implications for the formulation of our hypotheses are as follows:

- (a) we will adopt a relativist starting point, characteristic for the CDF, with respect to the superiority of any process modeling language,
- (b) we will consider both understandability and modifiability of process models as important sub-characteristics for maintainability,
- (c) we will consider modifications within the sequential and circumstantial spectrum, and
- (d) that we will build on viscosity to conjecture about declarative and imperative process modeling languages with respect to maintainability.

3 The Declarative-Imperative Spectrum

Given the insights from programming language research, this section analyzes to what extent an analogy can be established between procedural and declarative programming and respective approaches to process modeling. Section 3.1 elaborates on the difference between imperative and declarative programming and Section 3.2 discusses to what extent the distinction between sequential and circumstantial information is appropriate for process modeling.

3.1 Imperative versus Declarative Programming

Assuming that the reader has an intuitive understanding of what an imperative (or procedural) program is, we approach the topic from the declarative angle. According to Lloyd “declarative programming involves stating what is to be computed, but not necessarily how it is to be computed” [22]. Equivalently, in the terminology of Kowalski’s equation [23] ‘algorithm = logic + control’, it involves stating the logic of an algorithm (i.e. the knowledge to be used in problem solving), but not necessarily the control (i.e. the problem-solving strategies). While the logic component determines the meaning of an algorithm, the control component only affects its efficiency [23].

Roy and Haridi [24] suggest to use the concept of a *state* for defining the line between the two approaches more precisely. Declarative programming is often referred to as stateless programming as an evaluation works on partial data structures. In contrast to that, imperative programming is characterized as stateful programming [24]: a component’s result not only depends on its arguments, but also on an internal parameter, which is called its “state”. A state is a collection of values being intermediate results of a desired computation (at a specific point in time). Roy and Haridi [24] differentiate between implicit (declarative) state and explicit state. Implicit states only exist in the mind of the programmer without

requiring any support from the computation model. An explicit state in a procedure, in turn, is a state whose lifetime extends over more than one procedure call without being present in the procedure’s arguments. Explicit state is visible in both the program and the computation model.

3.2 Imperative versus Declarative Process Modeling

Process modeling is not concerned with programs, variables, and values, but aims at describing processes. In general, a *process* is a collection of observable actions, events, or changes of a collection of real and virtual objects. A *process modeling language* provides concepts for representing processes. Discussions of declarative versus imperative process modeling are scarce and so are precise distinctions. A description is given in Pestic’s PhD thesis [3, p.80]: “[Imperative] models take an ‘inside-to-outside’ approach: all execution alternatives are explicitly specified in the model and new alternatives must be explicitly added to the model. Declarative models take an ‘outside-to-inside’ approach: constraints implicitly specify execution alternatives as all alternatives that satisfy the constraints and adding new constraints usually means discarding some execution alternatives.” Below, we relate declarative and imperative modeling techniques to the notion of state.

An *imperative* process modeling language focuses on the aspect of *continuous* changes of the process’ objects which allows for two principal, dual views. The life of each object in the process can be described in terms of its *state space* by abstractly formulating the object’s *locations* in a real or virtual world and its possibilities to get from one location to another, i.e. state changes. The dual view is the *transition space* which abstractly formulates the distinct actions, events, and changes of the process and how these can possibly succeed each other. Based on topological considerations of Petri [25], Holt formally constructs a mathematical framework that relates state space and transition space and embeds it into the theory of *Petri nets* [26]. Holt deduces that Petri net places (or states in general) act as “grains in space” while Petri net transitions (or steps in general) act as “grains in time” providing dedicated concepts for structuring the spatial and the temporal aspect of a process. A directed flow-relation defines pre- and post-places of transitions, and corresponding pre- and post-transitions of places. Thus, in a Petri net model, beginning at any place (state) or transition, the modeler can choose and follow a *continuous* forward trajectory in the process behavior visiting more places (states of objects) and transitions. This interpretation positions Petri nets as a clear imperative process modeling language.

A *declarative* process modeling language focuses on the *logic* that governs the overall interplay of the actions and objects of a process. It provides concepts to describe *key qualities* of objects and actions, and how the key qualities of different objects and actions relate to each other in time and space. This relation can be arbitrary and needs not be continuous; it shall only describe the logic of the process. In this sense, a declarative language only describes *what* the essential characteristics of a process are while it is insensitive to *how* the process works. For instance, a possible key quality of a process can be that a specific action is “just being executed”. Formalizing this quality as a predicate ranging over a

set of actions, one can use the temporal logic LTL to model how executions of actions relate to each other over time. The logical implication thereby acts as the connective between cause and effect: Each action is executed a specific number of times (e.g. at least once, at most three times); the execution of one action requires a subsequent execution of some other action (at some point); the execution of two given actions is mutually exclusive; etc. Thereby state and step are not explicated in the model, but they are constructed when *interpreting* predicates and formulas. This kind of description relies on an *open-world assumption* leaving room for how the process' changes are continuously linked to each other. Any behavior that satisfies the model is a valid behavior of the process. This approach was formalized for modeling processes in the language ConDec [27].

Probably the most notable difference between imperative and declarative modeling is how a given behavior can be classified as satisfying a model or not. In an imperative model, the behavior must be reconstructible from the description by finding a continuous trajectory that looks exactly like the given behavior or corresponds to it in a *smooth* way. For instance, the linear runs of a Petri net are not explicitly visible in the net's structure, but states and steps can be mapped to places and transitions preserving predecessor and successor relations. In a declarative model, all requirements must be satisfied by the given behavior; there is no smooth correspondence required between behavior and model.

The reason for this difference between imperative and declarative modeling is the *degree* to which these paradigms make states and steps explicit. An imperative process model, like a Petri net, explicitly denotes states or steps or both and their direct predecessor-successor relations. Thus possible steps and successor states can be computed locally from a given state or step; runs can be constructed inductively. In a declarative model, like an LTL formula, states and steps are implicitly characterized by the predicates and the temporal constraints over these predicates. Any set of states and steps that are "sufficiently distinct" and relate to each other "sufficiently correct" are a valid interpretation of the model. This prohibits a construction of runs, but allows for characterizing states and steps as satisfying or not.

Despite these differences, declarative and imperative models can be precisely related to each other. While a direct transformation of declarative models into well-conceivable imperative models implies overhead, the resulting imperative model is operational and allows for executing declarative ones [27].

4 Changing Process Models

Having elaborated on the characteristics of imperative and declarative process modeling languages, this section aims to discuss the notion of process change in the context of both imperative and declarative process modeling languages.

Process change is the transformation of an initial process model S to a new process model S' by applying a set of change operations. A change operation modifies the initial process model by altering the set of activities and their order relations [28]. For imperative process modeling languages typical change

primitives are *add node*, *add edge*, *delete node* or *delete edge* [29]. In turn, for declarative process modeling languages like ConDec typical change primitives are *add activity*, *add constraint*, *delete activity* or *delete constraint* [30].

To conduct such a change to a process model the process designer 1.) needs to determine which changes have to be made to the model (i.e., to identify the necessary change operations) and 2.) apply the respective changes to the process model. Consequently, the effort needed to perform a particular process model change is on the one hand determined by the cognitive load to decide which changes have to be made to the model, which is a comprehension and sense-making task. On the other hand, the effort covers the number of edit operations required to conduct these changes, which is an action task. While the cognitive load is largely related to understandability issues of process models [4], the second issue can be related to the *viscosity* dimension of the CDF.

For the remainder, we explore to what extent repetitive viscosity and knock-on viscosity – the theoretical constructs from CDF – may apply to process model changes by discussing some well-chosen examples. Since we cannot hope to cover all existing process modeling notations, we restrict ourselves to two notations that seem characteristic for the poles of the imperative-declarative spectrum. For the imperative side we refer to workflow nets, a Petri net variant. For the declarative side, we refer to ConDec.

4.1 Repetitive Viscosity

The effect of repetitive viscosity can be measured in terms of the *process edit distance* of the initial process model S and the new process model S' . This is the minimum number of operations needed to transform S to S' [28]. Siddiqi defines repetitive viscosity exactly along these lines [21].

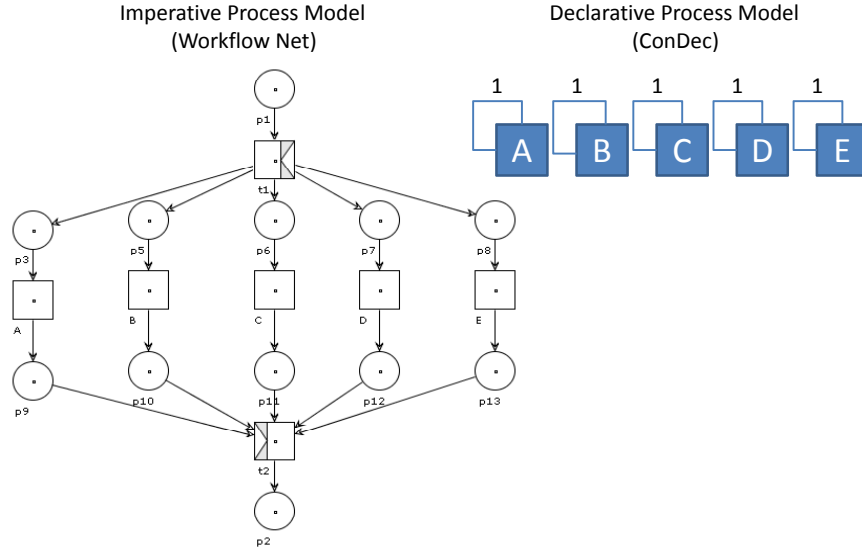


Fig. 2. Process Change: Imperative versus Declarative

Fig. 2 shows a simple business process comprising five distinct activities which are all to be executed exactly once in arbitrary order. The respective process is both modeled in an imperative manner using the workflow net notation and in a declarative manner using ConDec. Let us assume that this business process should be modified such that an additional activity **F** is inserted in parallel to the 5 already existing activities. To implement this change in the imperative process model seven change primitives are required (add node (3x), add add edge (4x)). The modification in the declarative model, in turn, requires 2 change primitives (add activity and add cardinality constraint). Consequently, repetitive viscosity for this particular change is lower for the declarative process model compared to its imperative counterpart. Consider another change scenario where an additional activity **G** should be inserted directly after activities **A-E**. For the imperative process model this change is relatively easy and only requires 4 change primitives (add node (2x) and add edge (2x)). For the declarative model this change turns out to be more complicated needing 7 change primitives (add activity, add cardinality constraint, and add precedence constraint (5x) – each of **A-E** “must precede **G**”). For this change repetitive viscosity is lower for the imperative process model compared to the declarative process model.

Obviously, tool support plays a fundamental role in reducing repetitive viscosity. To foster process changes and to hide the complexity from the end users adaptive process management systems like ADEPT combine change primitives to high-level change operations [29, 8]. Obviously, the usage of high-level change operations (often referred to as *change patterns*) reduces repetitive viscosity. However, our focus is to investigate the differences between different process modeling languages and we consequently consider change primitives only.

4.2 Knock-on Viscosity

Knock-on viscosity, in turn, becomes relevant when a particular process change entails further actions to restore consistency [6]. In the context of process model changes the deletion of an activity might require additional modifications to ensure data availability. In addition, changes of an activity interface might require changes of other activities as well. At this stage we abstract from the observation that higher coupling of activities in a model might result in a higher knock-on viscosity (see [11]), and focus on general language features.

Assume that activity **B** should be moved behind activity **C** in Fig. 2. In the ConDec model, this only involves adding a precedence constraint. In the workflow net, on the other hand, this operation requires several knock-on operations. Once **B** is connected with the path behind **C** there are different clean-up deletions to be made, e.g. deleting the place **p5** that served as a precondition to **B**. Apparently, ConDec is more robust to such clean-up actions that relate to knock-on viscosity.

4.3 Propositions

Altogether, we can conclude that the process edit distance implied by a change requirement is a major factor for viscosity. We have discussed that the set of

change operations offered by the modeling environment has a significant impact on this edit distance. At this stage, we have considered elementary change operations, e.g. adding or deleting nodes and edges in a workflow net. In addition, it might be reasonable to discuss types of changes within the sequential and circumstantial spectrum that Gilmore and Green find relevant for understanding [5]. Here, we call a change requirement as *sequential* if an activity has to be added or moved before or behind another activity. A *circumstantial* change requirement involves adding or moving an activity such that a general behavioral constraint is satisfied. We hypothesize that sequential changes to a workflow net are rather easy, but circumstantial ones are much more difficult. Indeed, it is often not easy to determine whether a circumstantial change (add A such that it is exclusive to C and concurrent to D) is possible at all. For ConDec we assume both types of operations being rather similar regarding the level of ease. This leads to our propositions:

- **H1:** Circumstantial changes are more difficult to apply to a business process modeled as a workflow net than sequential changes.
- **H2:** Circumstantial changes and sequential changes are equally difficult to apply to a business process modeled in ConDec.
- **H3:** Sequential changes are more difficult to apply in a business process modeled in ConDec than when modeled as a workflow net.
- **H4:** Circumstantial changes are more difficult to apply in a business process modeled as a workflow net than when modeled in ConDec.

Recall that earlier we distinguished two types of tasks related to process changes, i.e., those undertaken in a phase of sense-making (the cognitive tasks), and those undertaken in a phase of actually performing a number of change operations (action tasks). Since there is little understanding of the relative importance of both phases in applying process changes or their exact interaction, we aim to study these phases isolated from each other to the best of our ability.

We have discussed the aspects of understandability in an earlier paper [4], which arguably affect the first phase. Because our interest in this paper is with the second part of process change, the action task, we explicitly prefer to determine the presence or absence of the hypothesized differences in terms of measures that do not relate to time. After all, the overall modification time is affected by the duration of both phases. Rather, we prefer to rely on the *number of errors* that are induced by applying a change, assuming that a higher number of change primitives will inevitably lead to more mistakes.

Additionally, since it is impossible to rule out that sense-making is an important matter *during the action task*, we will prepare our experimental set-up such that we can control for differences in the understanding of the same business process modeled in the two notations under consideration. For example, we can use control questions on the understanding of the models before actual process changes are requested. By taking these measures, we hope to arrive at the best possible understanding of differences in the actual change effort that the different notations require from modelers in case of modifications.

5 Conclusion

In this paper, we presented a set of propositions that relate to the maintainability of process modeling languages. Specifically, these propositions focus on the distinction between declarative and imperative languages, formulating relative strengths and weaknesses of both paradigms. The most important theoretical foundation for these propositions is the cognitive dimensions framework including the results that are established for programming languages.

This paper is characterized by a number of limitations. First of all, there is a strong reliance on similarities between process modeling languages on the one hand and programming languages on the other. Differences between both ways of abstract expression may render some of our inferences untenable. At this point, however, we do not see a more suitable source of inspiration nor any strong counter arguments. Another limitation that is worth mentioning is that this paper primarily focuses on the effect the selection of the process modeling language has on maintainability of process models. However, resistance to change is also affected by the tools used for modification.

As follows from the nature of this paper, the next step is to challenge the propositions with an empirical investigation. We intend to develop a set of experiments that will involve human modelers carrying out a set of modification tasks on process models. Such tasks will involve both repetitive and knock-on viscosity including more and less declarative (imperative) languages. The cooperation of various academic partners facilitates extensive testing and replication of such experiments. Ideally, this empirical investigation will lead to an informed voice in the ongoing debate on the superiority of process modeling languages.

References

1. Svatoš, O.: Conceptual process modeling language: Regulative approach. In: 9th Undergraduate and Graduate Students eConf. and 14th Business & Government Executive Meeting on Innovative Cross-border eRegion, Univ. of Maribor (2007)
2. Boley, H., Meier, M., Moss, C., Richter, M.M., Voronkov, A.: Declarative and Procedural Paradigms - Do they Really Compete? In: PDK. LNCS 567, Springer (1991) 383–398
3. Pestic, M.: Constraint-Based Workflow Management Systems: Shifting Control to Users. PhD thesis, Eindhoven University of Technology (2008)
4. Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: The issue of understandability. In: BPMDS 2009 and EMMSAD 2009. LNBIP 29 (2009) 353–366
5. Gilmore, D.J., G.T.: Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies* **21**(1) (1984) 31–48
6. Green, T.R.G.: Cognitive dimensions of notations. In: Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V, Cambridge University Press (1989) 443–460
7. Mutschler, B., Reichert, M., Bumiller, J.: Unleashing the effectiveness of process-oriented information systems: Problem analysis, critical success factors, implications. *IEEE Trans. on Syst., Man, and Cybernetics (Part C)* **38**(3) (2008) 280–291

8. Reichert, M., Dadam, P.: ADEPT_{flex} – Supporting Dynamic Changes of Workflows Without Losing Control. *JIS* **10**(2) (1998) 93–129
9. Rinderle, S., Reichert, M., Dadam, P.: Correctness Criteria for Dynamic Changes in Workflow Systems – A Survey. *DKE* **50**(1) (2004) 9–34
10. Sarshar, K., Loos, P.: Comparing the Control-Flow of EPC and Petri Net from the End-User Perspective. In: Proc. BPM'05. LNCS 3649, Springer (2005) 434–439
11. Vanderfeesten, I., Reijers, H., van der Aalst, W.: Evaluating workflow process designs using cohesion and coupling metrics. *Computers in Industry* **59**(5) (2008) 420–437
12. Guceglioglu, A., Demirors, O.: Using Software Quality Characteristics to Measure Business Process Quality. In: Proc. BPM'05. LNCS 3649, Springer (2005) 374–379
13. Felleisen, M.: On the Expressive Power of Programming Languages. *Science of Computer Programming* **17**(1-3) (1991) 35–75
14. Prechelt, L.: An Empirical Comparison of Seven Programming Languages. *Computer* **33**(10) (2000) 23–29
15. Green, T.: Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology* **50** (1977) 93–109
16. Green, T.: Ifs and thens: Is nesting just for the birds? *Software Focus* **10**(5) (1980) 373–381
17. Green, T., Petre, M.: Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *J. of Vis. Lang. and Computing* **7**(2) (1996) 131–174
18. Blackwell, A.: Ten years of cognitive dimensions in visual languages and computing. *J. of Vis. Lang. and Computing* **17**(4) (2006) 285–287
19. Vanderfeesten, I., Reijers, H., Mendling, J., Aalst, W., Cardoso, J.: On a Quest for Good Process Models: The Cross-Connectivity Metric. In: Proc. CAiSE'08. LNCS 5074, Springer (2008) 480–494
20. Green, T., Blackwell, A.: A tutorial on cognitive dimensions. Available on-line at: <http://www.ndirect.co.uk/thomas.green/workStuff/Papers/index.html> (1998)
21. Siddiqi, J.I.A., Roast, C.R.: Viscosity as a metaphor for measuring modifiability. *IEE Proceedings - Software* **144**(4) (1997) 215–223
22. Lloyd, J.: Practical advantages of declarative programming. In: Joint Conference on Declarative Programming, GULP-PRODE'94. (1994)
23. Kowalski, R.: Algorithm = logic + control. *Commun. ACM* **22**(7) (1979) 424–436
24. Roy, P.V., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
25. Petri, C.A.: Concepts of net theory. In: Mathematical Foundations of Computer Science: Proc. of Symposium and Summer School, High Tatras, Sep. 3–8, 1973, Math. Inst. of the Slovak Acad. of Sciences (1973) 137–146
26. Holt, A.W.: A Mathematical Model of Continuous Discrete Behavior. Massachusetts Computer Associates, Inc. (Nov. 1980)
27. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: WS-FM. LNCS 4184, Springer (2006) 1–23
28. Li, C., Reichert, M., Wombacher, A.: On measuring process model similarity based on high-level change operations. In: ER'08. LNCS 5231, Springer (2008) 248–264
29. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data Knowl. Eng.* **66**(3) (2008) 438–466
30. Pesic, M., Schonenberg, M.H., Sidorova, N., van der Aalst, W.M.P.: Constraint-based workflow models: Change made easy. In: On the Move to Meaningful Internet Systems 2007. LNCS 4803, Springer (2007) 77–94