

Declarative versus Imperative Process Modeling Languages: The Issue of Understandability

Dirk Fahland¹, Daniel Lübke², Jan Mendling¹, Hajo Reijers³, Barbara Weber⁴,
Matthias Weidlich⁵, and Stefan Zugal⁴

¹ Humboldt-Universität zu Berlin, Germany

fahland@informatik.hu-berlin.de|jan.mendling@wiwi.hu-berlin.de

² Leibniz Universität Hannover, Germany

daniel.luebke@inf.uni-hannover.de

³ Eindhoven University of Technology, The Netherlands

h.a.reijers@tue.nl

⁴ University of Innsbruck, Austria

barbara.weber@uibk.ac.at|stefan.zugal@uibk.ac.at

⁵ Hasso-Plattner-Institute, University of Potsdam, Germany

matthias.weidlich@hpi.uni-potsdam.de

Abstract. Advantages and shortcomings of different process modeling languages are heavily debated, both in academia and industry, but little evidence is presented to support judgements. With this paper we aim to contribute to a more rigorous, theoretical discussion of the topic by drawing a link to well-established research on program comprehension. In particular, we focus on imperative and declarative techniques of modeling a process. Cognitive research has demonstrated that imperative programs deliver sequential information much better while declarative programs offer clear insight into circumstantial information. In this paper we show that in principle this argument can be transferred to respective features of process modeling languages. Our contribution is a pair of propositions that are routed in the cognitive dimensions framework. In future research, we aim to challenge these propositions by an experiment.

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

1 Introduction

At the present stage, formal properties of process modeling languages are quite well understood [1]. In contrast to these formal aspects, we know rather little about theoretical foundations that might support the superiority of one process modeling language in comparison to another one. There are several reasons why suitable theories are not yet in place for language design, most notably because the discipline is still rather young. Only little research has been conducted empirically in this area so far, e.g. [2, 3] relating model understanding to the modeling language and to model complexity.

The context for process modeling efforts is often that a model builder constructs a process model that aims to facilitate human understanding and communication among various stakeholders. This shows that the matter of understanding is well-suited to serve as a pillar on the quest for theories of process modeling language quality. Furthermore, insights from cognitive research on programming languages point to the fact that ‘design is redesign’ [4]: a computer program is not written sequentially; a programmer typically works on different chunks of the problem in an opportunistic order which requires a constant reinspection of the current work context. If process builders design their models in a similar fashion, understanding is an important quality factor for the modeler himself.

The lack of theories on modeling language quality with empirical support has contributed both to the continuous invention of new techniques and to the claims on the supposed superiority of such techniques. For instance, Nigam and Caswell introduce the OpS technique in which “the operational model is targeted at a business user and yet retains the formality needed for reasoning and, where applicable, automated implementation” implying that existing languages fall short on these characteristics [5]. In a Popkin white paper, Owen and Raj are less careful and claim a general superiority of BPMN over UML Activity Diagrams because “it offers a process flow modeling technique that is more conducive to the way business analysts model” and “its solid mathematical foundation is expressly designed to map to business execution languages, whereas UML is not” [6]. Smith and Fingar simply state in their book that “BPML is the language of choice for formalizing the expression, and execution, of collaborative interfaces” [7]. We do not want to judge on the correctness of these statements here, but rather emphasize that we currently lack theories to properly assess such claims.

Throughout this paper, we will discuss in how far insights from cognitive research on programming languages could be transferred to the process modeling domain. In particular, it is our aim to investigate the spectrum of imperative versus declarative process modeling languages, as this distinction can be considered as one of the most prominent for today’s modeling languages. For example, with respect to the recent development of ConDec (first published as “DecSer-Flow”), a declarative process modeling language, the first design criterion that is mentioned is that “the process models developed in the language must be understandable for end-users” [8, p.15]. While it is claimed that imperative (or procedural ¹) languages, in comparison, deliver larger and more complex process models, only anecdotal evidence is presented to support this. Also, in the practitioner community opinions are manifold about the advantages of declarative versus imperative languages to capture business processes, see for example [10–12]. These claims and discussions clearly point at the need for an objective, empirically founded validation of the presumed advantages of the different types of process modeling languages.

The contribution of this paper is that it presents a set of theoretically grounded propositions about the differences between imperative and declarative process

¹ Computer scientists prefer the term “procedural”; the term “imperative” is popular in other communities [9]. In this paper, we will be using the terms as synonyms.

modeling languages with respect understandability 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. To argue and support the hypotheses, this paper is structured as follows. Section 2 summarizes empirical findings and concepts from programming language research. 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. Section 5 concludes the paper and describes the empirical research agenda for validating the propositions.

2 Cognitive Research on Programming Languages

Various authors have noted the similarities between process models and software programs [13, 14]. For example, a software program is usually partitioned into modules or functions, which take in 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 their logic content. This stresses the importance of sense-making for both types of artifacts, both during the construction process and while updating such artifacts at a later stage.

While computer science is a relatively young field in relation to other engineering disciplines or the formal sciences, it has clearly a longer history than business process modeling. Therefore, it is worthwhile to reflect on the insights that are available with respect to the understanding of software code.

In the past, heated debates have taken place about the superiority of one programming language over the other with respect to expressiveness [15] or effectiveness [16], and such debates have extended to the issue of understandability. Edsger Dijkstra’s famous letter on the harmfulness of the `GOTO` statement, for instance, builds on the argument that “our powers to visualize how processes evolving in time are poorly developed” [17]. This made him dismissive of any higher level programming language supporting this construct. Another example is the development of visual programming languages, which have been claimed to be easier to understand than textual languages [18]. Finally, object-oriented programming languages have also been expected to foster understandability in comparison with more traditional languages, see e.g. [19].

During the 1970s and 1980s, alternative views were proposed on how programmers make sense of code as to provide a theoretical explanation of the impact of different programming languages on this process. One view is based on the idea of “cognitive restructuring”, in which problem-solving involves the access of information from both the world and memory (short- and long-term), and the restructuring of this information in working memory to provide a so-

lution. Therefore, languages from which information can be easily accessed and transferred to working memory will be easier to understand [20, 21].

An alternative view is that every programming language is translated into the same mental representation, and that comprehension performance reflects the extent to which the external program maps to people’s internal/cognitive representation primitives. This view is in line with certain theories on natural language processing [22] and forms the theoretical basis for experiments aimed at establishing people’s internal representations of computer programs [23, 24].

What has proven to be problematic with both these views is that they support the prediction that one programming language is easier or harder to understand than another in an absolute sense – whatever the exact aspect of the program that is studied. In work by Green [25, 26], and Gilmore and Green [27], however, it has both been postulated and empirically validated 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 is relatively easy to develop a mental picture of the control-flow for the same program. The implication of this view is that 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 was originally the basis for the “mental operations theory” [27], 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) [28, 29], which contains many different characteristics to distinguish notations from each other. Several of these dimensions directly matter to process modeling understanding, e.g. whether the model demands *hard mental operations* from the reader, whether there are *hidden dependencies* between notation elements, or whether changes can be applied locally (*viscosity*). The framework has been highly influential in language usability studies and over 50 publications have been devoted to its further development [30]. 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. While its application to business process models is, to our knowledge, limited to the work in [31], it seems to provide the strongest available theoretical foundation for our aims with this paper.

In particular, an important result that has been established in the development of the CDF relates to the difference between the tasks of looking for sequential and circumstantial information in a program. *Sequential* information explains how input conditions lead to a certain outcome. An example of looking for sequential information is: “In this program, after action X is performed, what might the next action be?”. Typically, one can distinguish between sequential information that relates to actions immediately *leading to* or *following from* a certain outcome. On the other hand, given a conclusion or outcome,

circumstantial information relates to the overall conditions that produced that outcome. An example of looking for *circumstantial* information is: “In this program, what combination of circumstances will cause action X to be performed?”. Circumstantial information may either relate to conditions that *have* or *have not* occurred. Empirical evidence is found to support 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 [25, 27]. The reverse is also true: Just as procedural languages tend to obscure circumstantial information, so do declarative languages tend to obscure sequential information. In other words, one “cannot simple-mindedly claim that procedural languages are easier or harder to read than declarative ones” [28].

The implication for this paper is (a) that we will adopt a similar relativist starting point for the formulation of our hypotheses and (b) that we will refine the distinction between sequential and circumstantial information within the context of process models.

3 The Declarative-Imperative Spectrum

Given the insights from programming language research, this section analyzes in how far 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; we discuss to which extent the distinction of sequential and circumstantial information is appropriate for process modeling thereafter. Section 3.3 illustrates the declarative-imperative spectrum with examples of process modeling languages.

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” [32]. Equivalently, in the terminology of Kowalski’s equation [33] ‘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 [33].

Roy and Haridi [34] 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 [34]: 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 [34] 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 Pesic’s PhD thesis [8, 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 [35], Holt [36] formally constructs a mathematical framework that relates state space and transition space and embeds it into the theory of *Petri nets* [1]. 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. Likewise, the modeler can follow a continuous backward trajectory to see the process behavior that leads to this place (state) or transition. 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 [37].

The probably 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 transitions explicit. An imperative process model like a Petri nets explicitly denotes states or transitions or both and their direct predecessor-successor relations. Thus enabled transitions and successor states can be computed locally from a given state or transition; runs can be constructed inductively. In a declarative model like an LTL formula states and transitions are implicitly characterized by the predicates and the temporal constraints over these predicates. Any set of states and transitions 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 transitions as satisfying or not.

Despite these differences, declarative and imperative models can be precisely related to each other. For instance, any LTL formula can equivalently be translated into a (finite) Büchi automaton [38]. The translation has the price of a technical overhead to express the genuine concepts of one language by the available concepts of another language. While this prohibits a direct transformation of declarative models into well-conceivable imperative models, the resulting imperative model is operational and allows for executing declarative ones [37].

3.3 A Characterization of Process Modeling Languages

As we stated in the previous section, process modeling languages differ with respect to the *degree* in which they make states and transitions explicit. This is

in line with Roy and Haridi’s [34] suggestion that “declarative” is no absolute property. The following languages lend themselves as evidence for this hypothesis as they position themselves in the imperative-declarative spectrum of process modeling languages. At the imperative end we position Petri nets, and LTL at the declarative end of the spectrum. Because of the large variety of process modeling languages, our list cannot be exhaustive.

Petri nets. We already illustrated the key concepts of imperative process modeling languages by the help of *Petri nets* which make state and transition explicit. A Petri net model of a process provides for each *atomic action* a dedicated transition and for each *atomic state* of a process resource a dedicated place. During modeling, one usually has to augment the model by further transitions and places to implement the desired process logic, e.g. loops, decisions, synchronization, etc. At any stage the modeler may mentally execute the process by placing mental tokens on the given places and mentally firing enabled transitions. These mental operations are supported by the continuous graph-based structure of the model that makes sequential information explicit as explained above. Several techniques like sub-nets transitions or patterns aid in structuring processes and making composite actions explicit.

Colored Petri nets [39] extend Petri nets by offering arbitrary values, objects, and structures to be passed through the net, instead of black tokens; these nets are used for modeling processes with data. Which colored tokens (values) are consumed, and how these are manipulated by firing transitions is specified in arc inscriptions and transition guards being *algebraic terms* with free variables. Thereby, the terms only denote how different colored tokens relate to each other allowing the transition to fire in many different modes. This adds circumstantial information to a transition which is positioned in a sequential context. The modeler has to mentally instantiate the arc inscriptions to get an explicit representation of the behavior. For larger pieces of continuous behavior, inscriptions of several transitions must be instantiated correspondingly.

Flow-based modeling languages like UML Activity Diagrams or BPMN materialize structuring techniques of Petri nets in dedicated modeling concepts. Besides different kinds of actions, these languages know *control-flow nodes* like *AND-split* and *XOR-join* to route control-flow between activities. *Event nodes* explicitly denote process instantiation, communication and termination. These modeling concepts offer a way to represent some of the key corresponding mental concepts of processes explicitly requiring fewer mental operations to understand the model.

The **Business Process Execution Language (BPEL)** has a *block-oriented* structure and provides even more specialized concepts for process modeling in a web service context. The block-oriented design allows to read a BPEL model like procedural program. But concepts like *exception handling*, *negative control-flow* and handling of *concurrent events* break the sequential nature of the process. The exact mechanics that coordinate normal process execution and exception handling etc. are not visible in the model, but hidden in the language. The modeler has to reconstruct them mentally to get a consistent image.

Scenario-based languages like Message Sequence Charts (UML Sequence Diagrams) and Life-Sequence Charts provide an explicit notion of behavior in terms of scenarios [40]. A *scenario* denotes a partial execution of the process as a partially ordered set of actions. A model is a set of scenarios sharing some actions. How actions of different scenarios relate to each other is not stated explicitly. Rather, a scenario’s structure and annotations describe how it can or cannot be extended by other scenarios. A scenario provides both, sequential and circumstantial information: It describes a continuous piece of behavior. At the same time, when asking “How to execute the last action of this scenario?”, it presents the partial answer “Execute all preceding actions of the scenario.”

The **Pockets of Flexibility** approach [41] combines imperative and declarative modeling elements in an integrated manner. Essentially, a *pocket of flexibility* constitutes a placeholder action in a flow-based process model; the pocket is dynamically refined to a flow-based process fragment at run-time. For each pocket declarative modeling constraints can be specified, which have to be obeyed upon refinement. A pocket introduces a region into a process model, where no explicit sequential information is available. The modeler has to link restricting constraints to the surrounding flow, and vice versa, when constructing the model.

TLA. The *Temporal Logic of Actions* (TLA) [42] allows to model process steps in terms of variable values in the current state and in the next state. Together with temporal operators like in LTL, TLA allows to model processes in terms of behavioral invariants as well as in terms of continuous changes.

ConDec. The process modeling language ConDec [37] formalizes key temporal relationships between executions of activities of a process in LTL patterns; e.g. the number of executions of an action or how two (or more) actions must or must not succeed each other. This makes some temporal concepts of process behavior explicit, similar to BPMN compared to Petri nets. The concepts of ConDec are stateless and give only circumstantial information for the (non-) executability of an action. The semantic domain of ConDec is limited to a specific, finite set of activities (out of which the process consists). Thus, the possibilities to relate different circumstances, like “executing action A” and “executing action B”, to each other are restricted. This eases a mental construction of continuous behavior that connects them.

LTL. The entire *Linear-Time Temporal Logic* (LTL) neither restricts process models nor the valid interpretations. The model may refer to further key qualities of a process like “availability of resource R”. Arbitrary circumstantial information can be constructed with the logical connectives, specifically the implication to relate cause and effect, and the temporal operators *always* (φ holds), *eventually* (φ holds), and *until* (φ holds until ψ holds). The *next* operator allows to express sequential information as it denotes a specific situation holding in the next state.

The languages which we have just presented highlight some points in the imperative-declarative spectrum of process modeling languages. The concepts range from an explicit notion of state and step to an explicit notion of process logic. Our list shows that an explicit notion of step does not exclude an explicit

notion of logic as most languages provide concepts for both. An important observation on our examples is that if process logic is explicated, an explicit notion of state is put in relation to that logic, and vice versa. The information that is conveyed by one explicit notion is *relative* to the information conveyed by other explicit notions. The reason for this relativity roots in the following observation.

Every explicit notion conveys some implicit, *hidden information*. Whatever is not explicated is implicit in the model as it can and must be inferred. Whenever step and logic are explicated together in some way, the implied, hidden information of one concept must be consistent with explicit information of the other concept. Picking up the analogy to programs, the process control (states and steps) must enact the process logic, and the process logic must be implemented in the process control. A relative interpretation of the language concepts provides the freedom for a consistent combination of both.

Our illustration of the imperative-declarative spectrum of process modeling languages shows that there are no predetermined points for combining imperative and declarative concepts, but that languages contain both in varying degrees.

4 Propositions

As stated in the introduction, it is our purpose to formulate a set of propositions that can be used as a basis to evaluate the comprehensiveness of process models specified in an imperative or declarative spirit. At this point, we have explored two important elements for this purpose. In the first place, we presented the CDF as the most plausible and dominant theory for sense-making of information artifacts in Section 2. Most notably, it stresses the task-notation relationship, e.g. in the *hard mental operations* and *hidden dependency* dimensions. This has provided us with a relativist viewpoint on the superiority of process modeling techniques – it is the match between the task and the language that will determine the overall effectiveness, not the technique in absolute terms. Also, the important concepts of finding *sequential* and *circumstantial* information give a strong clue to what types of tasks may give a better match with imperative or declarative process models.

Secondly, we reviewed the distinction between declarative and imperative process modeling languages in Section 3. We argued that the more a process modeling language emphasizes *states* and *transitions*, the more imperative it can be regarded. Similarly, the more a process modeling language relies on providing the mere requirements on acceptable behavior, the more declarative it is. Inherent to these views is our acceptance that the distinction between declarative and imperative process modeling languages is not a binary one. By combining the two elements, we arrive at the two following main propositions:

- P1** Given two semantically equivalent process models, establishing sequential information will be easier on the basis of the model that is created with the process modeling language that is relatively more imperative in nature.
- P2** Given two process models, establishing circumstantial information will be easier on the basis of the model that is created with the process modeling

language that is relatively more declarative in nature. Establishing circumstantial information will be easier on the basis of a declarative process model than with an imperative process model.

The reasoning for these propositions can be directly related to the *hard mental operations* and *hidden dependencies* dimensions. Sequence is a hidden dependency from the perspective of a declarative language and requires hard mental operations to construct it. An imperative language, on the other hand, is demanding in terms of circumstantial information because it is hidden and mentally hard to reconstruct. Specifically, we would expect that these propositions hold whether ease of understanding is measured in terms of *accuracy* or *speed*, cf. operationalizations of these notions in [4].

Finally, consistent with the CDF, we would expect these propositions to hold both when subjects have *direct* access to the process model and when they have to establish this information on *recall*, i.e. the memorization of a process model they have seen earlier. Remember that the CDF refutes the idea that people shape a similar problem situation into the same mental model, regardless of the form in which it is presented to them.

5 Conclusion

In this paper, we presented a set of propositions that relate to the understandability 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. Also, it is argued that any actual process modeling language finds itself somewhere on the spectrum from a less to a more imperative (declarative) nature. An analysis of existing process modeling languages is provided to support this argument.

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. Note that it can be argued that the issue of understandability may be even more important in the domain of process modeling than that of programming. After all, not only designers are reading process models but end users too – which is unusual for computer programs. Furthermore, we have focused exclusively on the issue of understanding but other quality aspects may be equally important. If design is redesign, as argued in this paper, not only understanding but also *ease of change* is important. There are respective cognitive dimensions that need to be discussed for process modeling notations, in particular, *viscosity* (ease of local change) and *premature commitment*.

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 ex-

periments that will involve human modelers to carry out a set of tasks that involve sense-making of a set of process models. Such tasks will be characterized by establishing both sequential and circumstantial information and including more and less declarative (imperative) languages. The cooperation of various academic partners in this endeavor 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. Reisig, W., Rozenberg, G., eds.: Lectures on Petri Nets I: Basic Models, Advances in Petri Nets. LNCS **1491** (1998)
2. Recker, J., Dreiling, A.: Does it matter which process modelling language we teach or use? an experimental study on understanding process modelling languages without formal education. In Toleman, M., Cater-Steel, A., Roberts, D., eds.: 18th Australasian Conference on Information Systems (2007) 356–366
3. Mendling, J., Reijers, H., Cardoso, J.: What makes process models understandable? In: Proc. BPM’07. LNCS **4714** (2007) 48–63
4. Gilmore, D.J., Green, T.R.G.: Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies* **21**(1) (1984) 31–48
5. Nigam, A., Caswell, N.: Business artifacts: An approach to operational specification. *IBM Systems Journal* **42**(3) (2004) 428–445
6. Owen, M., Raj, J.: BPMN and Business Process Management: Introduction to the New Business Process Modeling Standard. Technical report, Popkin, <http://whitepaper.techweb.com/cmptechweb/search/viewabstract/71> (2003)
7. Smith, H., Fingar, P.: Business Process Management: The Third Wave. (2003)
8. Pesic, M.: Constraint-Based Workflow Management Systems: Shifting Control to Users. PhD thesis, Eindhoven University of Technology (2008)
9. Boley, H.: Declarative and Procedural Paradigms - Do They Really Compete? In: Proc. PDK; 91, Springer (1991) 383–385
10. Korhonen, J.: Evolution of agile enterprise architecture. <http://blog.jannekorhonen.fi/?p=11>. Retrieved 10 Feb. 2009 (Apr. 2006)
11. Goldberg, L.: Seven deadly sins of business rules. <http://www.bpminstitute.org/articles/article/article/seven-deadly-sins.html>. Retrieved 10 Feb. 2009 (Sept. 2007)
12. McGregor, M.: Procedure vs. process. http://www.it-director.com/blogs/Mark_McGregor/2009/1/procedure_vs_process.html. Retrieved 10 Feb. 2009 (Jan. 2009)
13. Vanderfeesten, I., Reijers, H., van der Aalst, W.: Evaluating workflow process designs using cohesion and coupling metrics. *Comp. in Ind.* **59**(5) (2008) 420–437
14. Guceglioglu, A., Demirors, O.: Using Software Quality Characteristics to Measure Business Process Quality. In: Proc. BPM’05, Springer (2005) 374–379
15. Felleisen, M.: On the Expressive Power of Programming Languages. *Science of Computer Programming* **17**(1-3) (1991) 35–75
16. Prechelt, L.: An Empirical Comparison of Seven Programming Languages. *Computer* (2000) 23–29
17. Dijkstra, E.: Letters to the editor: go to statement considered harmful. *Communications of the ACM* **11**(3) (1968) 147–148
18. Glinert, E.: Nontextual programming environments. In: *Visual Programming Systems*. Prentice-Hall (1990) 144–230

19. Wiedenbeck, S., Ramalingam, V., Sarasamma, S., Corritore, C.: A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers* **11**(3) (1999) 255–282
20. Meyer, R.: Comprehension as affected by the structure of the problem representation. *Memory & Cognition* **4**(3) (1976) 249–255
21. Shneiderman, B., Mayer, R.: Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming* **8**(3) (1979) 219–238
22. Fodor, J., Bever, T., Garrett, M.: *The Psychology of Language: An Introduction to Psycholinguistics and Generative Grammar*. McGraw-Hill Companies (1974)
23. McKeithen, K., Reitman, J., Rueter, H., Hirtle, S.: Knowledge organization and skill differences in computer programmers. *Cogn. Psych.* **13**(3) (1981) 307–325
24. Adelson, B.: Problem solving and the development of abstract categories in programming languages. *Memory & Cognition* **9**(4) (1981) 422–33
25. Green, T.: Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology* **50** (1977) 93–109
26. Green, T.: Ifs and thens: Is nesting just for the birds? *Software Focus* **10**(5) (1980) 373–381
27. Gilmore, D., Green, T.: Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies* **21**(1) (1984) 31–48
28. Green, T.: Cognitive dimensions of notations. In Sutcliffe, A., Macaulay, L., eds.: *People and Computers V, Proceedings*. (1989) 443–460
29. Green, T., Petre, M.: Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *J. Vis. Lang. Computing* **7**(2) (1996) 131–174
30. Blackwell, A.: Ten years of cognitive dimensions in visual languages and computing. *J. Vis. Lang. Computing* **17**(4) (2006) 285–287
31. Vanderfeesten, I., Reijers, H., Mendling, J., Aalst, W., Cardoso, J.: On a Quest for Good Process Models: The Cross-Connectivity Metric. *LNCS* **5074** (2008) 480–494
32. Lloyd, J.: Practical advantages of declarative programming. In: *Joint Conference on Declarative Programming, GULP-PRODE'94*. (1994)
33. Kowalski, R.: Algorithm = logic + control. *Commun. ACM* **22**(7) (1979) 424–436
34. Roy, P.V., Haridi, S.: *Concepts, Techniques, and Models of Computer Programming*. MIT Press (2004)
35. 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
36. Holt, A.W.: *A Mathematical Model of Continuous Discrete Behavior*. Massachusetts Computer Associates, Inc. (Nov. 1980)
37. van der Aalst, W.M.P., Pesic, M.: DecSerFlow: Towards a truly declarative service flow language. In: *Proceedings of WS-FM*. *LNCS* **4184** (2006) 1–23
38. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design* **1**(2/3) (1992) 275–288
39. Jensen, K.: *Coloured Petri Nets*. Springer-Verlag (1992)
40. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. *Form. Methods Syst. Des.* **19**(1) (2001) 45–80
41. Sadiq, S., Sadiq, W., Orłowska, M.: A Framework for Constraint Specification and Validation in Flexible Workflows. *Information Systems* **30**(5) (2005) 349 – 378
42. Lamport, L.: The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* **16**(3) (1994) 872–923