

# Software Process Validation: Comparing Process and Practice Models<sup>1</sup>

Aldo de Moor<sup>1</sup> and Harry Delugach<sup>2</sup>

<sup>1</sup> VUB STARLab, Pleinlaan 2, 1050 Brussels, Belgium  
ademoor@vub.ac.be

<sup>2</sup> Department of Computer Science, University of Alabama in Huntsville,  
Huntsville, AL 35899, USA  
delugach@cs.uah.edu

**Abstract.** To assure the quality of software processes, models play an important role. Process models represent the officially sanctioned software development processes in the organization. Although important, they are not sufficient, since the practices of software developers often differ considerably from the official process. Practice models, describing the way software development is really done, are an important source of information for validating the software process. Using conceptual graph theory, we present a formal method for representing and comparing process and practice models in various combinations. The method allows for differences between these models to be easily detected. Software developers, such as managers or engineers, can then interpret these differences to make recommendations for software process improvement.

## 1 Introduction

Software development is a complex collaborative process. Organizations, whether following official standards or not, have their own preferred software development processes, often uniquely tailored to the projects and organizational characteristics at hand [4][8]. To capture these official processes, organizations develop their own *process models*, which guide, support and advise software developers by prescribing what activities and steps they are to follow in the production of quality software [6].

Although process models are important in providing overall guidance, they are seldom followed exactly. Developers often have different, personal views on what roles they are supposed to play. Furthermore, as contingencies and breakdowns occur in actual systems development, software engineering is critically dependent on the unique abilities of the creative people involved in those processes [12]. Thus, although a process focus is important for obtaining coherence in the organization, it is

---

<sup>1</sup> A. de Moor and H. Delugach (2006). Software Process Validation: Comparing Process and Practice Models. In *Proc. of the 11<sup>th</sup> International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2006), Luxembourg, June 5-6, CaiSE 2006 Workshop Proceedings*, Presses Universitaires de Namur.

the practice of people in the organization that brings the process to life [1]. Practices allow people to learn, innovate, and deal with contingencies; they are thus an essential part of software development, especially in an agile or distributed environment, where contingencies and breakdowns need to be dealt with all the time.

In this paper, we focus on exploring the differences between process and practice models, with the aim of providing useful insights to improve both software engineering process models and their resulting practice. We use the term *software process* for the normative, prescribed way of doing things in a software development organization. The intended software process is described by a *process model*. We use the term *practice* to refer to what developers are actually doing. In comparing process and practice, it is essential to *detect* and *compare* the differences between process model and practice. How large are the differences? How significant are the differences? In which steps of the development process do they occur? According to whose viewpoint are they (most) visible?

When comparing process and practice, however, there is an extra complexity. A practice, like the process, also has a model; in fact, each individual in the organization will have his or her own mental model of how software practices *really* work, for example, in order to circumvent – in their eyes – unnecessary bureaucracy or ineffective procedures. Developers often adopt ways of working (around) based on many instances of similar problems. Their individual *practice models* thereby consolidate both the individual lessons learned from many cases and possibly some short-cuts. These models are therefore an important intermediate unit of analysis. Since informal models are often error-prone, subject to bias, and especially hard to check and compare, we have chosen to formally represent process and practice models, since formalization makes the comparison process more effective and efficient. We can also use the same techniques to compare different process models. Our formalism of choice is conceptual graph theory [7], since, first, it allows complex specifications to be intuitively captured and, second, it contains powerful operations (beyond the scope of this paper) for these captured specifications to be compared and reasoned about.

In Sect. 2 we present our conceptual model of software process validation. Sect. 3 presents our validation method. In Sect. 4, we show the results of applying our method to a real-world aerospace software case where adequate software process validation is obviously important. Sect. 5 contains our conclusion.

## 2 Software Process Validation

Software development already includes a number of formal models for specifying requirements, analysis, architectural design, and so on. Analyzing and comparing these models is an integral activity in the process and hence already familiar (in principle at least) to participants. It is therefore not a large leap to consider formal models in order to validate the software development process itself. Our starting point is the *intended process*, the way in which the management of the organization envisions that the software process should be run. However, the intended process is often very different from the actual work going on. How does one validate the process? That is, how does one determine that the process is the right one? We emphasize that by *validation* we mean any approach that can evaluate differences between models; we do not imply any judgement here about whether one model is to be favored over another.

Fig. 1 presents our conceptual model of software process validation. We distinguish four types of validation: (i) first-hand observation, (ii) process-model validation, (iii) practice-model validation, and (iv) model comparison.

A classic kind of validation is the manager walking around the office, and observing how people work, without any explicit representation. This type of validation is an example of *first-hand observation*. In large, complex and spatially distributed organizations, or in organizations with rapidly evolving flexible processes, such ad hoc validation does not suffice anymore.

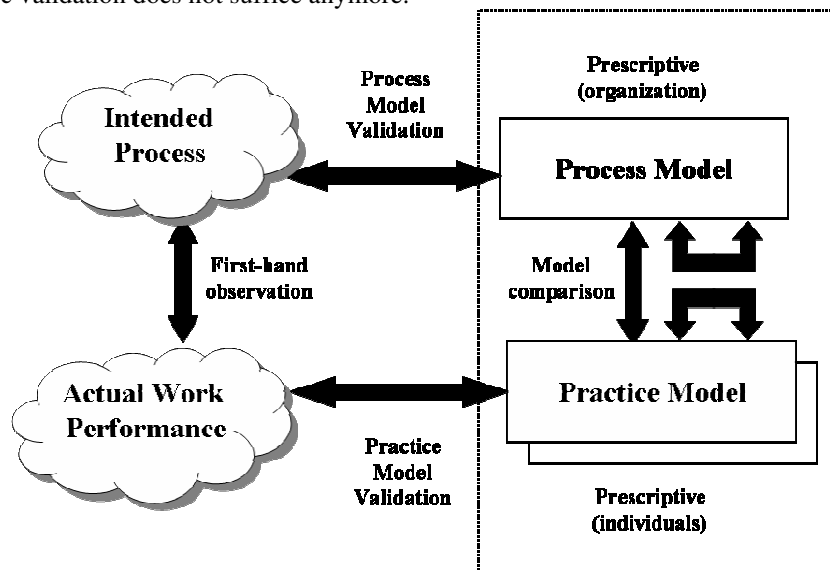


Fig. 1. A Conceptual Model of Software Process Validation.

A second type of validation should compare management intentions with the process model, which we call *process model validation*. A process model provides the organizational blueprint of the way business processes should be conducted. To some extent, the official description is an (informal) meta-model of this software development process; however, the actual intentions of management are often different from this official process model (or set of models, if there is more than one). Management needs are therefore often transformed in unintended ways during the specification process of the process model. One typical way to facilitate process model validation is to have external consultants conducting discussion sessions with management, based on analysis of the organizational process model with industry standards, such as CMM-process targets.

A third type of validation is between actual work performance and the individual practice models of staff members. This *practice model validation* can be useful to allow employees to improve their skills, by forcing them to reflect upon the difference between what they think is their optimal work performance, and what they are actually doing. This individual learning process can be promoted by such techniques as video-taping, logging command sequences, and so on.

In this article, we focus on a fourth type of software process validation: *model comparison* between the official process model and an individual practice model, or between two process or practice models. The coordination of many participants with very diverse interests and responsibilities means that the theoretical practices (i.e. process model) and the individual real practices (i.e. practice models) are often far removed from each other [11]. When old patterns of past experience no longer apply, new organizational patterns need to be developed and absorbed by staff [5].

The question now becomes how to efficiently facilitate such model comparison? Much work has focused on informal model comparison; e.g., the so-called Knowledge Workshops, which bring together experts and practitioners who, in a facilitated and structured way, discuss and capture learning and understanding [10]. Although certainly useful, such informal approaches are expensive, error-prone, and can easily lead to organizational memory-loss, since evaluation results are often hard to re-use or expand in future sessions. Advantages of formal methods include that (i) they reduce ambiguity, (ii) they can serve as a contract and help resolve conflicts over the interpretation, allowing for the reasoning about properties, and (iii) they are a prerequisite for many sophisticated analysis techniques [9]. In the remainder of this paper, we introduce a formal method that we have developed for comparing process and practice models, taking into account the non-trivial difficulties of putting it to use in the real world of time-constrained software professionals.

### 3 A Formal Method for Software Process Validation

At the heart of the method is a top-level software process validation ontology. Core ontological concepts are workflow, step, and activity. A key feature of a software development process is that it consists of (possibly concurrent) *workflows* (e.g. requirements acquisition) that lead to the final deployment of software. Within each workflow, there will usually be a set of *steps*, within each step a set of *activities*.

Our basic unit of analysis is an *activity* in which an *object* is produced. All *activities* require that at least one *organizational actor* initiates, executes, or evaluates them. We call these initiations, executions, or evaluations *control processes* (sometimes shortened to *controls*). Associated with these control processes are *norms*. These norms denote whether a particular actor must, may, or may not control a particular activity, in other words, whether that organizational actor has a responsibility, a privilege, or a prohibition with respect to that control process. The norms are labelled by their particular *deontic effect*, for which we use the terms “Required” for responsibilities, “Permitted” for privileges, and “Prohibited” for prohibitions. This normative approach draws from the RENISYS method for modelling workflow changes acceptable to a community [3].

Based on these concepts, the ontology includes a number of *activity patterns*. Definitional patterns represent concepts and relationships that must be present. For example, every activity must have some kind of control associated with it. Schematic patterns represent typical or expected concepts and relationships that are often present, but are not required. For example, not every activity will require an input object, so the relationships involving an input object are necessarily schematic rather than definitional. We use both definitions and schemas to organize the acquisition of models

The steps of the method are outlined in Table 1. The technical details of this method, including its use of conceptual graph theory as the underlying formal knowledge representation and reasoning formalism have been explained in [2]. Here we only illustrate the method by showing an example taken from a case we studied of software engineering processes in a large aerospace organization.

**Table 1.** A Formal Method for Software Process Validation

<b>•Create activity patterns for the models</b>
<b>•Create two models</b>
○ Models can be either process or practice models
○ Identify model steps
○ Build models by instantiating or specialising general activity patterns for each step
<b>•Compare the models</b>
○ Identify activities in models, and delineate steps in a sequence
○ Compare steps between the models
○ Create difference graphs
<b>•Interpret differences between the models</b>
○ Present difference graphs to software professionals
○ Adapt process and/or practice models
○ Launch change program

#### 4 Applying the Method: Aerospace Software Engineering

Our example is based on a detailed study of a small-sized internal software development group that develops and maintains aerospace software. This particular group is characterized as small (10-20 persons), leading to multiple roles per person, with little duplication or crosstraining of roles, occasional role re-assignment, and (we discovered) implicit accumulating adaptations of the official process in practice.

Software development in this group is project-based. We compared its process and practice models in a project with non-mission critical environmental control system software. We also compared the non-mission critical project's process model with the process model for mission critical propulsion system software developed in another project, but led by the same manager. Space does not permit us to show our complete model of a software process; we focused on one small part: namely, this organization's activity of creating and approving changes to the requirements.

First, we gathered information from official documents and interviews with a key manager familiar with both the published process and actual practice. After eliciting many partial graphs, we were able to construct a process and a practice model. At first (human) sight, the graphs looked rather similar. However, finding the differences between the graphs forms the heart of our comparing process and practice models. Using conceptual graph theory, we computed the difference between two graphs as

the sequence of difference operations that will transform one graph into the other. Subsequently, the differences were highlighted visually in what we call *difference graphs*. In [2], we explained the algorithms and illustrated them by analyzing the difference graphs of the process and practice model for the non-critical environmental system software development. In this paper, we will not focus on the technical procedure, but give another example: the comparison of the non-mission critical environmental system software process model (Proj. A) with that of the mission critical propulsion system software (Proj. B). Fig. 2 shows Project A's view on the differences.

The dark lines delineate steps which are identified by simple rules (omitted here) for defining a temporal ordering. The highlighted parts are where the two models differ. Here there are significant differences between the graphs; in fact, there is an extra step where the Software Review Board (SRB) evaluates a change request in Project-A's but not in Project-B's process model. Also some steps have roughly the same structure in both graphs, but the particular object concepts are different.

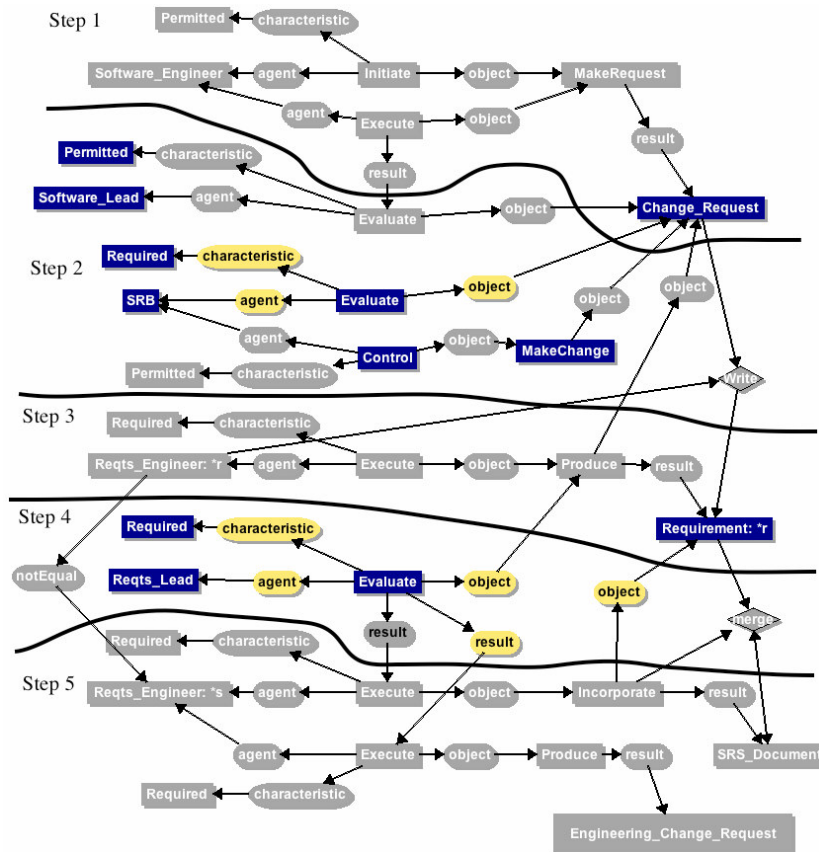


Fig. 2. Difference graph for Project-A and B's process models (Project-A's perspective)

## 5 Interpretation of key differences between the process graphs

We used the actual graph models in consultation with the manager to analyze the

differences between the projects. Even though the same person was a manager for both projects, our meeting quickly identified areas that he felt deserved attention to improve process quality. More than once he remarked, “I never thought of that!” as we talked through the differences. While not rigorous, our experience was encouraging, suggesting the considerable potential for this technique. Some observations were:

In the first step, in Project-A, the activity is the same as in Project-B, but the result of the activity is a **Change\_Request** in Project-A, and a **System\_Note** in Project-B (not shown in this graph). Further, note that in Project-A only one **Software\_Lead** does the evaluation and is permitted (but not required) to do it, whereas in Project-B (shown in the other difference graph), a whole team (the so-called System Note Review Team) is involved and is required to do so. The (required) involvement of more people might mean that the **System\_Note** has more value to the organization.

Furthering the analysis of the first step, however, we see that in Project-A there is an entire extra evaluation step performed by the **SRB** (which is a team) that is required to evaluate the **Change\_Request**. This would tend to contradict our previous conclusion since now multiple people and two roles are involved in the evaluation process. We would expect, however, that more overall effort would be expended for Project-B since it is the mission-critical one. Still, in that project there is only one role involved in the evaluation, whereas Project-A gets two independent evaluations. To reconcile this contradiction, we would need more information such as team size, the length of time spent on and methodology of evaluation, etc.

Focusing now on Step 2 which is centered around the **Make\_Change** activity in Project-A and the **Produce** activity in Project-B, one difference is that in Project-A **Make\_Change** affects the result of a previous activity (**Make\_Request**), whereas in Project-B the **Produce** activity follows from the previous activity and has a new result **Requirements\_Change\_Notice**. Furthermore, in Project-A, the **SRB** is both permitted to change the **Change\_Request** itself and required to evaluate the change, but in Project-B, the software engineer is required to produce the **Requirements\_Change\_Notice**, and a (possibly different) set of software engineers is required to evaluate the **Requirements\_Change\_Notice**. It makes sense that in mission-critical Project-B there would be additional people involved in the evaluation, but the graph does not show whether the initial producer of the change is also involved in the evaluation, more information is needed here.

There is a terminology difference in Step 3 between **Requirement** in Project-A and **Release\_Requirements\_Change\_Notice** in Project-B. This may be just a matter of usage, but it may also indicate that Project-A has underspecified the lifecycle status of the requirements or that Project-B has overstated it.

We see the requirements engineer’s execution in Step 5 is a direct result of the requirements lead’s evaluation in step 4; there is no explicit initiation. We have learnt that practitioners often combine initiation and execution in eliciting models. It seems plausible to assume that if some organizational actor executes an activity that has no explicit initiation, then that actor is also the initiator. This may not always be desired for quality control purposes that favor a separation of responsibilities.

Note that for each difference, we were able to identify a course of action or suggested further examination. This gives us confidence that the technique helps software development practitioners wanting to improve process quality.

## 6 Conclusion

Software process validation is an essential but complex part of the systems devel-

opment process, requiring a subtle mix of specific validation processes. In this article, we focused on one particular validation process: comparing two process models. Differences between these models can be an important source of creativity for identifying problems and opportunities for improvement of current work regulations and practices and ensuring quality control.

The method we presented makes use of conceptual graph theory, a powerful knowledge formalism that is well suited to elicit, represent, reason about, and compare the complex knowledge structures such as process and practice models. There are additional conceptual graph features (e.g., type/relation hierarchies, context, actors) that lend themselves to even more interesting evaluation. Since there is a cost in creating the conceptual graphs, we are working on tools and techniques that can improve our efficiency in creating them from documents and interviews.

By using the formalism for *identifying* the differences, while leaving the *interpretation* of these differences to the domain experts such as software engineers and managers, we think we have established a balance between computational power and human intelligence and experience. As our experience with the aerospace software process shows, the method-under-construction was already found useful by its intended users, in the sense of providing new insights at relatively low cost. Future work will measure the cost of model creation vs. the benefit of difference evaluation.

Analytical approaches often use informal techniques, making it hard to see the bigger picture in different stories of organizational actors. In software engineering, however, formal methods are common, but these often focus only on the improvement of low-level software execution processes. Thus, a major gap exists with respect to the evaluation of organizational governance and practices at the business process and workflow level. Our method for software process validation aims to fill this gap.

**Acknowledgments:** The research in this paper has been partially supported by the EU Leonardo da Vinci CODRIVE project (BE/04/B/F/PP-144.339) and DIP project (EU-FP6 507483) .

## References

1. Brown, J. S. and Duguid, P., 2000. *The Social Life of Information*, Harvard Business School Press, Cambridge, MA.
2. Delugach, H. and De Moor, A., 2005. Difference Graphs. In *Contributions to ICCS 2005*, Kassel, Germany, 2005, pp.41-53.
3. de Moor, A. and Jeusfeld, M.A., 2001. Making Workflow Change Acceptable, *Requirements Engineering*, 6(2):75-96.
4. Jones, C., 2003. Variations in Software Development Practices, *IEEE Software*, 20(6), 22-27.
5. Kurtz, C. F. and Snowden, D. J., 2003. The New Dynamics of Strategy: Sense-Making in a Complex and Complicated World, *IBM Systems Journal*, 42(3), 462-483.
6. Rolland, C., Souveyet, C. and Moreno, M., 1995. An Approach for Defining Ways-of-Working, *Information Systems*, 20(4), 337-359.
7. Sowa, J. F., 1984. *Conceptual Structures : Information Processing in Mind and Machine*, Addison-Wesley, Reading, Mass.
8. ter Hofstede, A. H. M. and Verhoef, T. F., 1997. On the Feasibility of Situational Method Engineering, *Information Systems*, 22(6/7), 401-422.
9. van der Aalst, W. M. P., 1998. Three Good Reasons for Using a Petri-Net-Based Workflow Management System. In: Yates, J. (Ed.), *Information and Process Integration in Enterprises: Rethinking Documents*, Kluwer, Boston, 161-182.

10. von Krogh, G., Nonaka, I. and Aben, M., 2001. Making the Most of Your Company's Knowledge: A Strategic Framework, *Long Range Planning*, 34, 421-439.
11. Winograd, T., 1995. From Programming Environments to Environments for Designing, *Communications of the ACM*, 38(6), 65-74.
12. Winograd, T. and Flores, F., 1986. *Understanding Computers and Cognition : a New Foundation for Design*, Ablex Pub. Corp., Norwood, N.J.