A Heuristic Method for Business Process Model Evaluation

Volker Gruhn and Ralf Laue

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

Abstract. In this paper, we present a heuristic approach for finding errors and possible improvements in business process models. First, we translate the information that is included in a model into a set of Prolog facts. We then search for patterns which are related to a violation of the soundness property, bad modeling style or otherwise give raise to the assumption that the model should be improved. By testing our approach on a large repository of real-world models, we found that the heuristic approach identifies violations of the soundness property almost as accurate as model-checkers that explore the state space of all possible executions of the model. Other than these tools, our approach never ran into state-space explosion problems. Furthermore, our pattern system can also detect patterns for bad modeling style which can help to improve the quality of the models.

1 Introduction

In the past years, numerous static code analysis tools have been developed for finding software bugs automatically. They analyze the source code statically, i.e. without actually executing the code. There exist several good tools that are matured to be useful in production environments. Examples of such tools are Splint (`www.splint.org`), JLint (`jlint.sourceforge.net`) or FindBugs (`findbugs.sourceforge.net`).

These static analysis tools use very different technologies for localizing errors in the code, including dataflow analysis, theorem proving and model checking [1]. All the tools and technologies have in common that they use some kind of **heuristics** in order to find possible problems in the code. This means that it can happen that such a tool reports a warning for code that is in fact correct (*false positive*) as well as the tool can fail to warn about an actual error (*false negative*).

It is important to mention that static analysis can be applied not only for locating bugs, but also for detecting so-called "bad code smells" like violations of coding conventions or code duplication. This means that not only erroneous code but also code that is hard to read or hard to maintain can be located.

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

In this paper, we show how some ideas behind such static analysis tools can be transferred to the area of business process modeling.

2 The EPC Notation

We have developed our approach using the business process modeling language Event-Driven Process Chains [2]. There are two reasons for this choice: The first reason is that this modeling language is very widespread (at least this is the case for Germany) and we have been able to collect a large repository of models. Secondly, the EPC notation is a rather simple notation which is made up of the basic modeling elements that can be found in more expressive languages like BPMN or YAWL as well. These basic constructs will be introduced in the remainder of this section.

EPCs consist of functions (activities which need to be executed, depicted as rounded boxes), events (pre- and postconditions before / after a function is executed, depicted as hexagons) and connectors (which can split or join the flow of control between the elements). Arcs between these elements represent the control flow. The connectors are used to model parallel and alternative executions. There are two kinds of connectors: Splits have one incoming and at least two outgoing arcs, joins have at least two incoming arcs and one outgoing arc.

AND-connectors (depicted as $\textcircled{\scriptsize\wedge}$) are used to model parallel execution. After an AND-split, the elements on all outgoing arcs have to be executed in parallel. An AND-join connector waits until all parallel control flows that have been started are finished.

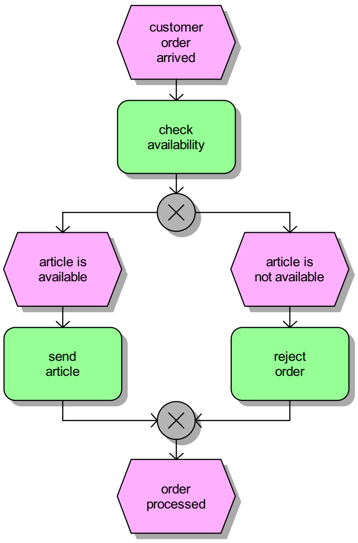

Fig. 1. Simple Business Process modeled as EPC

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

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

Fig. 1 shows a simple business process modeled as EPC diagram. The meaning of this model is as follows: When a request from a customer arrives, the availability of the product has to be checked. If it is available, the item will be sent; otherwise the customer will get a negative reply.

3 Our General Approach: Pattern Matching

The key idea of our approach is to search for patterns of "bad modeling". For the purpose of finding such patterns in a model, we used logic programming with Prolog. The XML serialization of the model has been translated into a set of facts in a Prolog program (as described in [3]). Each function, event, connector and arc in the model is translated into one Prolog fact.

Furthermore, we have constructed Prolog rules that help us to specify the patterns we are looking for.

We started with defining some rules describing the basic terminology, for example by specifying that a connector is called a split if it has one incoming and more than one outgoing arc or by recursively defining what we want to call a path from some node to another.

Secondly, we defined some important relations between split and join nodes that we can use to define the patterns. The most important definition is the one for the relation `match(S,J)`. It means that a split S corresponds to a join J such that S branches the flow of control into several paths that are later merged by a join J. As we cannot assume that splits and joins are properly nested, this definition is the prerequisite for finding patterns that are related to control-flow errors in arbitrary structured models. We have defined the Prolog clause `match(S,J)` such that S is a split, J is a join and there are two paths from S to J whose only common elements are S and J.

Furthermore, we defined exits from and entries into a control block between a split S and a join J for which `match(S,J)` holds. For example, an exit from such a structure is defined such that there is a path from S to an end event (i.e. an event without outgoing arc) that does not pass J or a path from S to S that does not pass J. In Fig. 2, model (c) is the only one that has an exit from the control block between s and j.

By a systematic analysis of all possible patterns of matching split-join pairs with or without "exits" and "entries", we developed a set of 12 patterns that are indicators for control-flow errors.

Here, we will discuss one such pattern which is a good example of the heuristic nature of our approach: One of our rules states that we suppose the existence of

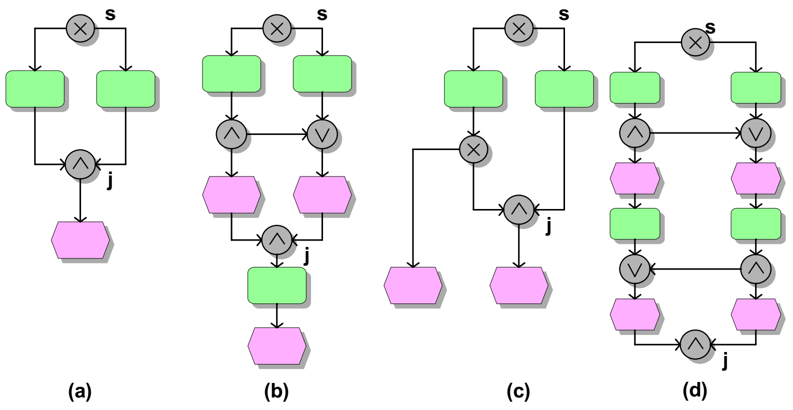

Fig. 2. The rightmost model fragment does not have a deadlock, the others have

a deadlock if there is an (X)OR-split s and an AND-join j such that `match(s,j)` holds, regardless of whether there are exits or entries between the split and the join. In most real-world models, such a pattern is indeed related to a deadlock, some cases are shown in Fig. 2 (a)-(c).

In these models, the outgoing arcs from the XOR-split s are later joined by an AND-join j. While only *one* outgoing flow from the XOR-split will be processed, the AND-join has to wait until *all* incoming arcs have been completed - a typical deadlock situation.

However, in some rare occasions (as the one shown in Fig. 2 (d)) the pattern is found in a model that in fact does not have a deadlock.

The existence of such rare cases where our rules would give the wrong information is inherent to the heuristic idea of our approach. It was not our goal to make our pattern catalogue as complete as possible. In the same way as it is known from code analysis tools, we had to find a balance between accuracy (i.e. prevention of false positives and false negatives) and speed of execution[1].

In the next sections, we will show which kind of problems can be located by applying different kinds of patterns.

4 Control-Flow Errors

The most important correctness criterion for business process models is the soundness property, originally introduced by van der Aalst for workflow nets [4,5] and later adapted to the EPC notation [2,6].

For a business process model to be sound, three properties are required:

1. In every state that is reachable from a start state, there must be the possibility to reach a final state *(option to complete)*.

[1] In fact, the case shown in Fig. 2 (d) is even considered by the latest version of our Prolog rule set: The rules will produce an "error" alert for the models (a)-(c) and a "possible error" alert for model (d).

2. If a state has no subsequent state (according to the transition relation that defines the precise semantics), then only events without outgoing arcs (end events) must be marked as being "active" in this state *(proper completion).*
3. There is no element of the model that is never processed in any execution of the model *(no needless elements).*

Violations of the soundness criterion usually indicate an error in the model. Therefore, 12 out of the 24 patterns we have defined so far aim to locate control-flow errors that can lead to a violation of the soundness property. An example (the combination of an (X)OR-split and an AND-join) has already been discussed in Sect. 3.

5 Comprehensibility and Style

Correctness (in terms of the soundness property) is not the only quality requirement for business process models: One of the main purposes of such models is to be used as a language in a discussion between humans. In particular, the models can serve as a bridge between the stakeholders in a software development project. They are formal enough to serve the demands of software developers but easy enough to be understood by business experts as well.

For this purpose, business process models should be as easy as possible to comprehend. If there is a choice among different modeling elements to express the same situation, the most comprehensible alternative should be used.

For example, in some cases it is possible to replace an OR-connector by an AND- or XOR-connector which describes the situation much better (for a human reader) without changing the semantics of the model.

As example, take a control block where an AND-split starts two paths that are executed in parallel. Formally, it is correct to join both paths using an OR-join. The meaning of the OR-join is to wait for all paths that have been started as a prerequisite for transferring control to its outgoing path. This means that in Fig. 3 (a), the OR-join acts exactly as an AND-join. While it would not make a difference for the actual meaning of the model, the readability of the model can be improved by substituting the OR-join by an AND-join. The same idea can be applied for the other model fragments in Fig. 3: In model (b) and (c), the

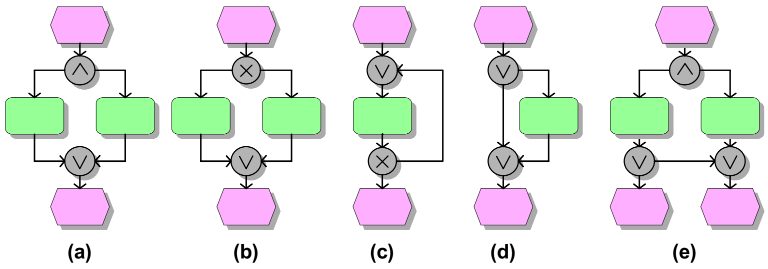

Fig. 3. Models with OR-connectors that should be replaced

OR-join should be replaced by an XOR-join. In model (d), a pair of XOR-split and XOR-join should be used to model the fact that an activity can either be skipped (by taking the left path) or executed (by taking the right one). Fig. 3 (e) shows another situation where an OR-split can be replaced by an XOR-split. With an XOR-split, the mental load for the reader of the model is reduced: He or she has not to consider the case that both outgoing arcs of the split are followed.

Using Prolog rules that specify the above patterns (and a few more that are not described here due to space restrictions), we are able to advice the modeler to change the model in order to improve its readability.

An organization can add own style rules, for example in order to enforce the policy that all pairs of splits and joins have to be properly nested (which is not required by the notation, but sometimes desirable).

6 Pragmatic Errors

So far, we have discussed "technical" errors (like deadlocks) and the readability of the model. It is almost impossible to validate automatically whether the model really represents the real business process without building an ontology of the application domain (see for example [7,8]) which is usually far too time-consuming and expensive.

There are however, some patterns that "look like" the model does not reflect the real business process. In such a situation, the modeler can be informed to double-check the questionable part of the model. We have discussed one such case (that we call Partial Redo pattern) in [9]. Another (simpler) pattern is shown in Fig. 4: After a function has been performed, an XOR-connector splits the flow of control and exactly one of two alternative events happens. However, afterwards both paths are joined again and the future execution of the process is the same regardless of which event actually occurred. But why does the model show that two different events can happen if the execution actually does not care whether the right event or the left event occurred? In some cases, this might make sense for documentation purposes, but it is also not unlikely that the modeler did forget something in the model.

In our survey of real-world models we found examples where this pattern indeed was the result of an error that had to be corrected, but of course we

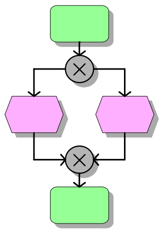

Fig. 4. Why is the alternative between two events modeled if these events do not have an effect on future execution?

also found models in which such a pattern just helps to understand the possible progress of the business process. The false warnings produced by the heuristics are less problematic, compared to the reward when a correct warning helps to locate an actual modeling error.

7 Validation

We searched for the patterns described above (and some more that cannot be described in detail due to space restrictions) in a repository of 984 models. Those models have been collected from 130 sources. These sources can be categorized as follows:

- 531 models from the SAP R/3 reference model, a widespread business reference model
- 112 models from 31 bachelor and diploma thesises
- 25 models from 7 PhD thesises
- 13 models from 2 technical manuals
- 82 models from 48 published scientific papers
- 12 models from 6 university lecture notes
- 4 models from sample solutions to university examination questions
- 88 models from 11 real-world projects
- 88 models from 7 textbooks
- 29 models from 14 other sources

Among the models in our repository, there is a great variation in size of the models, purpose of modeling, business domain and experience of the modelers. For this reason, we think that the models represent a reasonable good sample of real-world models.

Before doing the soundness analysis with these tools, we reduced all models using soundness-preserving reduction rules as described in [10,6]. 534 of the models have been reduced to a single node which means that they are sound. An analysis was necessary for the remaining 450 models.

In order to compare our heuristic results to the results of exact soundness analysis, we selected three well-known open-source tools that check business process models for the soundness property: *EPCTools*, the *ProM plugin for EPC soundness analysis* and the *YAWL Editor*. These tools have to compute the state space of all possible executions of the model. Because this state space can contain a great number of states, such tools can suffer from the so-called state-space explosion - a situation in which there is not enough available memory to store all possible states or the computation can not be done in a reasonable time.

All these tools run as a Java program. We executed the tools on an Intel Core2 Duo CPU running at a speed of 3 GHz. By starting the Java virtual machine with the option `-Xmx1536m`, we allowed a heap size of 1.5 GB to be used.

EPCTools [11,12] calculates a fixed-point semantics for a model. If such a fixed-point semantics exists, a temporal model-checker is used by *EPCTools* for deciding about the soundness property. For the majority of models, an analysis result was given in a few seconds. There was, however, one model for which the analysis took more than 10 minutes; this model was validated in 63 minutes. For 7 models, *EPCTools* had to stop the computation because of an *Out of Memory* error.

The *ProM plugin for EPC soundness analysis* [13] uses the semantics defined by Mendling [6] for constructing a transition system for the model. For 31 models, ProM failed to deliver a result because of an *Out of Memory* error. For 5 models, the computation took more than 10 minutes, the longest computation time was 26 minutes.

The third tool, *YAWL Editor* [14,15], originally has been constructed for analyzing YAWL models. While the mapping of EPC modeling elements to YAWL is straightforward, there is an important difference between EPC and YAWL: YAWL does not support process models with more than one start node. In order to avoid the problems that arise from the different instantiation semantics for EPC and YAWL models [16], we considered only those non-trivial 203 models for which the EPC model has exactly one start event (after applying the reduction rules). YAWL Editor has a built-in restriction that stops the execution of the analysis if the state-space exceeds 10,000 states. This is necessary, because the YAWL semantics allows an infinite state space [14]. This restriction was enforced for 13 models, meaning that no analysis result for them was available. While the computation was very fast for the majority of the models (133 have been analyzed in less than 1 second), there were also some that took much longer: For 8 models, the computation took more than 10 minutes. Two models could not be analyzed within one hour; the longest computation time was 390 minutes.

For all tools which we have tested, some models turned out to be “hard cases” where a Java heap space of 1.5 GB and a time of one hour was not enough to judge about the soundness property by exploring the state space. EPCTools had 8 such “hard cases” (out of 450), the ProM plugin for EPC soundness analysis 31 (out of 450) and YAWL Editor 15 (out of 203).

In contrast, our Prolog-based tool needed only 65 seconds for analyzing *all* models from the repository. This time included searches for 12 patterns that are related to control-flow errors as well as searches for more 12 patterns that are related to bad modeling style or suspiciously looking modeling elements.

After doing the analysis with the different tools, we compared the results. Because of subtle differences among the tools when it comes to defining the semantics of the OR-join [17,18], there were a few differences in the results of the tools. The analysis of these differences will be the subject of another paper; here it is sufficient to say that in cases of differences among the tools we looked at the model and selected the result that complied with the intuitive understanding of its semantics.

The comparison between our heuristic results and the exact results showed that the heuristics worked almost as good as the state-space exploring tools: For all models which have been categorized as not being sound by the “exact” tools, our Prolog program also found at least one pattern that (likely) shows a violation of the soundness property, i.e. we have had no “false negatives”. On the other hand, our program warned about a possible soundness violation for exactly one model that turned out to be sound, i.e. we have had only one "false positive". It is worth mentioning that the model, for which this false positive occurred, was taken from [19] where it has been published as an example for bad modeling that should be improved.

YAWL Editor also allows checking whether an OR-join should be replaced (as this was the case for the model fragments in Fig. 3 (a)-(c)) which would not affect soundness, but can be considered as an improvement in the modeling style as discussed in Sect. 5. Our heuristics missed just one case where such a substitution was possible (false negative), but did not produce any incorrect warnings (false positives).

8 Related Work

Logic-based and pattern-matching approaches have been used in many published approaches for finding modeling errors. Their main application area is the detection of syntactical errors and inconsistencies within one model or between different models [20]. Our approach adds one more perspective by also detecting control-flow errors (like deadlocks) and even pragmatic issues.

Störrle [21] showed that a representation of models as logic facts can be very useful for querying model repositories as well.

ArgoUML [22] is an excellent example of a user-friendly modeling tool that runs tests in background in order to give the modeler feedback about possible improvements. So-called "design critics" inform the modeler about possible problems. The user is allowed to create own design critics. The most design critics currently implemented in ArgoUML either work on a rather syntactical level (i.e. they check consistency requirements and constraints that have to be followed according to the UML standard) or test whether modeling style conventions [23] are followed.

Work on error patterns for business process models has been done by different authors ([24,25,26,27]. As none of these pattern systems considered all three types of connectors that can occur in EPC models (OR, AND and XOR), they are of limited use for the assessment of business process models in languages in which all these connectors can be found.

The approach by Mendling [6] which applies reduction rules for finding errors considers all kinds of connectors. It is able to find a great part of errors in EPC models. Therefore, we used the reduction rules given in [6] as one starting point for creating our patterns. However, Mendling does not lay importance on the completeness of the rules; he uses reduction rules mainly for simplifying a model before using a state-space exploring algorithm for validating the model.

All the pattern systems mentioned above (which all share a set of basic common patterns) are reflected in our rules for detecting control-flow errors. Our definition of matching splits and joins, which is one of the most fundamental rules of our rule system, was inspired by the use of traps and handles in workflow nets for finding control-flow errors [28,5].

9 Conclusions and Directions for Further Research

In our analysis of a large number of business process models, we found that our pattern-based approach performed almost as good as tools that apply model-checking when it comes to detect soundness violations and OR-joins that should be replaced by XOR- or AND-joins. An advantage of our approach is that it produced a result very fast while the other tools suffered from the symptoms of state-space explosion and failed to deliver a result for some models. Furthermore, we have also patterns for hard-to-read parts of the model and "suspiciously looking" parts of the model that might indicate a pragmatic error even if the model is sound.

Using Prolog, the patterns can be specified very easily, and it is possible to add new patterns (for example for applying organization-wide style conventions) very quickly. However, our pattern-based approach does not necessarily have to be used with Prolog or another logic-based language. We have already implemented a pattern-finding algorithm in the open source Eclipse-based modeling tool *bflow**[2] using the languages *oAW Check* and *XTend* from the openArchitectureWare model management framework [29]. With this implementation, *bflow** gives the modeler immediate feedback about possible modeling problems.

Currently, we are working on an implementation using the query language BPMN-Q [30]. This will allow us to apply our approach to BPMN models.

One future direction of research is to consider more sophisticated modeling elements (like exceptions or cancellation) that exist in languages like BPMN or YAWL. This will allow us to deal with more complex patterns like the ones we have discussed in [9].

We are also researching problems that can be found by analyzing the textual description of events and functions. We are already able to find some problems this way. For example, if an OR-split is followed by both an event and its negation (as "article is (not) available)" in Fig. 1), it is very likely that the OR-split has to be replaced by an XOR-split, because both events cannot happen together.

References

1. Rutar, N., Almazan, C.B., Foster, J.S.: A comparison of bug finding tools for java. In: ISSRE, pp. 245–256 (2004)
2. van der Aalst, W.M.: Formalization and verification of event-driven process chains. Information & Software Technology 41, 639–650 (1999)
3. Gruhn, V., Laue, R.: Checking properties of business process models with logic programming. In: Augusto, J.C., Barjis, J., Ultes-Nitsche, U. (eds.) MSVVEIS, pp. 84–93. INSTICC Press (2007)
4. van der Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997)
5. van der Aalst, W.M.P.: Structural characterizations of sound workflow nets. Computing Science Reports/23 (1996)

[2] http://www.bflow.org

6. Mendling, J.: Detection and Prediction of Errors in EPC Business Process Models. PhD thesis, Vienna University of Economics and Business Administration (2007)
7. Fillies, C., Weichhardt, F.: Towards the corporate semantic process web. In: Berliner XML Tage, pp. 78–90 (2003)
8. Thomas, O., Fellmann, M.: Semantic EPC: Enhancing process modeling using ontology languages. In: Hepp, M., Hinkelmann, K., Karagiannis, D., Klein, R., Stojanovic, N. (eds.) SBPM. CEUR Workshop Proceedings, vol. 251. CEUR-WS.org (2007)
9. Gruhn, V., Laue, R.: Good and bad excuses for unstructured business process models. In: Proceedings of 12th European Conference on Pattern Languages of Programs (EuroPLoP 2007) (2007)
10. van Dongen, B.F., van der Aalst, W.M.P., Verbeek, H.M.W.: Verification of EPCs: Using reduction rules and Petri nets. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 372–386. Springer, Heidelberg (2005)
11. Cuntz, N., Kindler, E.: On the semantics of EPCs: Efficient calculation and simulation. In: EPK 2004: Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings, pp. 7–26 (2004)
12. Cuntz, N., Freiheit, J., Kindler, E.: On the Semantics of EPCs: Faster calculation for EPCs with small state spaces. In: EPK 2005, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, pp. 7–23 (2005)
13. Barborka, P., Helm, L., Köldorfer, G., Mendling, J., Neumann, G., van Dongen, B.F., Verbeek, E., van der Aalst, W.M.P.: Integration of EPC-related tools with ProM. In: Nüttgens, M., Rump, F.J., Mendling, J. (eds.) EPK. CEUR Workshop Proceedings, vol. 224, pp. 105–120. CEUR-WS.org (2006)
14. Wynn, M.T.: Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-joins. PhD thesis, Queensland University of Technology Brisbane, Australia (2006)
15. Wynn, M.T., Verbeek, H., van der Aalst, W.M.P., Edmond, D.: Business process verification - finally a reality! Business Process Management Journal 15, 74–92 (2009)
16. Decker, G., Mendling, J.: Instantiation semantics for process models. In: Proceedings of the 6th International Conference on Business Process Management, Milan, Italy (2008)
17. van der Aalst, W.M.P., Desel, J., Kindler, E.: On the semantics of EPCs: A vicious circle. In: EPK 2004, Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, pp. 71–79 (2002)
18. Dumas, M., Grosskopf, A., Hettel, T., Wynn, M.: Semantics of BPMN process models with or-joins. Technical Report Preprint 7261, Queensland University of Technology, Brisbane (2007)
19. Mendling, J., Reijers, H.A., van der Aalst, W.M.P.: Seven process modeling guidelines (7pmg). Technical Report QUT ePrints, Report 12340, Queensland University of Technology (2008)
20. Finkelstein, A.C.W., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multiperspective specifications. IEEE Trans. Softw. Eng. 20, 569–578 (1994)
21. Störrle, H.: A prolog-based approach to representing and querying software engineering models. In: Cox, P.T., Fish, A., Howse, J. (eds.) VLL. CEUR Workshop Proceedings, vol. 274, pp. 71–83. CEUR-WS.org (2007)
22. Robbins, J.E., Redmiles, D.F.: Cognitive support, UML adherence, and XMI interchange in Argo/UML. Information & Software Technology 42, 79–89 (2000)

23. Ambler, S.W.: The Elements of UML Style. Cambridge University Press, Cambridge (2003)
24. Onoda, S., Ikkai, Y., Kobayashi, T., Komoda, N.: Definition of deadlock patterns for business processes workflow models. In: Proceedings of the 32nd Annual Hawaii International Conference on System Sciences, vol. 5, p. 5065. IEEE Computer Society, Los Alamitos (1999)
25. Koehler, J., Vanhatalo, J.: Process anti-patterns: How to avoid the common traps of business process modeling, part 1 - modelling control flow. IBM WebSphere Developer Technical Journal (2007)
26. Liu, R., Kumar, A.: An analysis and taxonomy of unstructured workflows. In: Business Process Management, pp. 268–284 (2005)
27. Smith, G.: Improving process model quality to drive BPM project success (2008), `http://www.bpm.com/improving-process-model-quality-to-drive-bpm-project-success.html` (accessed November 1, 2008)
28. van der Aalst, W.M.P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
29. Kühne, S., Kern, H., Gruhn, V., Laue, R.: Business process modelling with continuous validation. In: Pautasso, C., Koehler, J. (eds.) MDE4BPM 2008 – 1st International Workshop on Model-Driven Engineering for Business Process Management (2008)
30. Awad, A.: BPMN-Q: A language to query business processes. In: Reichert, M., Strecker, S., Turowski, K. (eds.) EMISA, GI. LNI, vol. P-119, pp. 115–128 (2007)