Business Process Modeling Flexibility: A Formal Interpretation

Anila Mjeda[1], Andrew Butterfield[2] and John Noll[1,3]
[1]*Lero, The Irish Software Research Centre, U. of Limerick, Ireland*
[2]*School of Computer Science and Statistics, Trinity College Dublin, Ireland*
[3]*University of East London, U.K.*

Keywords: Business Process Modelling, Flexible Interpretation, Formal Semantics, Unifying Theories of Programming.

Abstract: Domain experts from both the software and business process modelling domains concur on the importance of having concurring and co-supportive business and software development processes. This is especially important for organisations that develop software for regulated domains where the software development processes need to abide by the requirements of the domain-specific quality assurance standards. In practice, even when following quite mature development processes to develop high assurance systems, software development is a complex activity that typically involves frequent deviations and requires considerable context-sensitive flexibility. We took a business process modelling notation called PML that was specifically designed to be lightweight and allow flexibility, and developed formal semantics for it. PML supports a range of context-sensitive interpretations, from an open-to-interpretation guide for intended behaviour, to requiring a precise order in which tasks must occur. We are using Unifying Theories of Programming (UTP) to model this range of semantic interpretations and the paper presents a high-level view of our formal semantics for PML. We provide examples that illustrate the need for flexibility and how formal semantics can be used to analyse the equivalence of, or refinement between, strict, flexible, and weak semantics. The formal semantics are intended as the basis for tool support for process analysis and have applications in organisations that operate in regulated domains, covering such areas as the certification process for medical device software.

1 INTRODUCTION

Software development is a complex activity that requires frequent adaptation from the developers to cope with external and internal forces such as changing requirements, evolving design knowledge, and failures.

Software development in regulated domains such as medical devices presents additional complexity due to the restrictions on the software process imposed by regulations: depending on the potential impact of device failure, regulations stipulate that certain processes must be followed to manage risks and improve safety.

Organizations must meticulously document how their processes comply with these regulations and this heightens the need of synchronicity/synergy between their business and software development processes. Indeed, domain experts from both the software science and business modelling domains concur on the pivotal role of these two processes' mutual support (Aguilar-Saven, 2004).

In this context, it becomes important to be able to document an organization's development process in sufficient detail to satisfy regulatory constraints, while still allowing developers as much flexibility as possible within those constraints.

Modelling both business processes and the (regulated) software development processes in the same notation can provide a much-needed *lingua franca* which enables a common understanding of the constraints, dependencies and synergies of an organisation's business and software development processes.

This, for example, could facilitate using the same modelling notation both to satisfy regulators and to inform developers as to what tasks are required by regulations; this ensures that changes to the process are reflected in both presentations.

To satisfy these requirements, we took a business process modelling notation called "PML" that was specifically designed to be lightweight and allow flexibility. We then developed formal semantics for three levels of interpretation for PML:

1. Strict - the specified control flow and pre- and post-conditions dictate the exact order and con-

Mjeda, A., Butterfield, A. and Noll, J.
Business Process Modeling Flexibility: A Formal Interpretation.
DOI: 10.5220/0007577104650472
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 465-472
ISBN: 978-989-758-358-2

ditions for enactment of the process.

2. Flexible - the specified control flow must be enacted as written, except that sequential steps can be re-ordered as long as pre- and post-conditions are met, and concurrent activities need not all complete before subsequent activities can start.
3. Weak - the specified control flow can be ignored: steps can be performed in any order as long as their pre-conditions are met. Steps can also be skipped if their post-conditions are met.

This defines a hierarchy of flexibility from none to maximum. The implication is that a (business or software) process specification could be written that has a strict interpretation that satisfies regulatory constraints. If the weak interpretation can be shown to be equivalent, through comparison of the two interpretations' semantics, then the regulators' need for compliance, and developers' need for flexibility, are satisfied with a single process and process description.

The remainder of this paper is organized as follows. In the next section, we introduce PML and provide an example that will serve to illustrate the need for flexibility and, later, how formal semantics can be used to analyze equivalence of strict, flexible, and weak interpretations. Following that we describe the formal semantics and how they are derived using UTP. Finally, we discuss related work, then conclude with implications and future directions.

2 PML AND PROCESS FLEXIBILITY

The Process Modelling Language (PML) (Atkinson et al., 2007) is a shared-state concurrent imperative language that has been designed to model organizational/business processes. PML models a process as a collection of atomic tasks, each of which `requires` resources to start, and `provides` resources when it completes. PML uses four constructs to model process flow: `sequence`, `iteration`, `selection` and `branch`:

1. Sequence: Models a series of tasks to be performed in the specified order:

```
sequence {
  action A {}
  action B {}
}
```

2. Iteration: Models a series of tasks to be performed repeatedly, where the body is implicitly interpreted as a `sequence`:

```
iteration {
  action A {}
  action B {}
}
```

3. Selection: Models a set of tasks from which only one can be chosen to be performed:

```
selection {
  action choice_A { }
  action choice_B { }
}
```

4. Branch: Models a set of concurrent tasks, all of which have to be performed:

```
branch {
  action path_A { }
  action path_B { }
}
```

The `iteration` and `branch` constructs in PML are underspecified by design and behave somewhat unusually. For instance, the `iteration` construct has no explicit termination condition. PML acknowledges the flexible nature of processes and leaves the decision to the `agent` responsible for enacting the process to decide when the loop should terminate.

This means that a given `iteration` construct can be interpreted in at least two different ways:

- The `agents` (people) enacting the business process use their judgement to determine when the actions in the body of the `iteration` have been repeated enough.
- The availability of *resources* in the system serves as loop control. An `action` in a PML model can have `required` resources, that must be available before the action and begin, and `provided` resources, that become available when the action completes. The `iteration` stops when the `required` resources of the action following the iteration are available, and the `required` resources of the first action in the body of the iteration are *not* available.

Similarly flexible is the behaviour of the `branch` construct. The decision on when to proceed beyond the branch join point is left unspecified, and thus is left to the judgement of the `agent` enacting the process model.

In essence, depending on the interpretation, the trace (enactment history) of a specific process model can consist of: 1. an iteration of the non-deterministic choice of actions whose resources are available; 2. a sequence of actions that is governed solely by when the required resources become available; or 3. a precise pre-defined sequence of actions which deadlock if the required resources are not available.

```
process collect_signatures {
  action obtain_PM_sig {
    requires { document_approval_form
        }
    provides { document_approval_form
        .pm_signed }
    agent { ProjectManager }
  }
  action obtain_dept_head_sig {
    requires { document_approval_form
        }
    provides { document_approval_form
        .dept_head_signed }
    agent { DepartmentHead }
  }
  action obtain_dir_sig {
    requires { document_approval_form
        }
    provides { document_approval_form
        .director_signed }
    agent { DivisionDirector }
  }
  action obtain_VP_sig {
    requires { document_approval_form
        }
    provides { document_approval_form
        .VP_eng_signed }
    agent { VPEngineering }
  }
  action distribute_document {
    requires { document &&
        document_approval_form.
        pm_signed &&
        document_approval_form.
        dept_head_signed &&
        document_approval_form.
        director_signed &&
        document_approval_form.
        VP_eng_signed }
    provides { document.distributed }
    agent { Author }
  }
}
```

Figure 1: PML specification of Collect Signatures process. Note that the body of a `process` is implicitly interpreted as a `sequence`.

2.1 Initial Semantic Considerations

The flexibility of different semantic interpretations of process terms allows PML to model a process at different levels of granularity and at each of those levels to cater for context-sensitive interpretations.

To illustrate the attractiveness of the idea, let us consider a *CollectSignatures* workflow that is required in a document change approval process. An organization must put such a process in place in order to comply with US FDA regulation Title 21 Subchapter H Part 820 "Medical Devices Quality Management Regulation," (Food and Drug Administration, United States Department of Health and Human Services, 2018b) which states in sub-part 820.20:

> Each manufacturer shall designate an individual(s) to review for adequacy and approve prior to issuance all documents established to meet the requirements of this part. The approval, including the date and signature of the individual(s) approving the document, shall be documented.

In our example, suppose the document must be approved by the Project Manager, Department Head, Division Director, and Vice President of Engineering. This would be documented on a document change approval form that collects signatures from these individuals. A PML specification for this process is shown in Fig. 1.

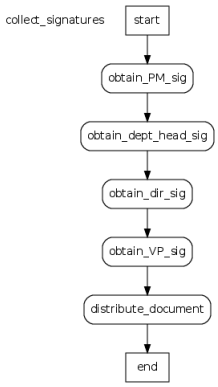

Figure 2: Strict interpretation of Collect Signatures process.

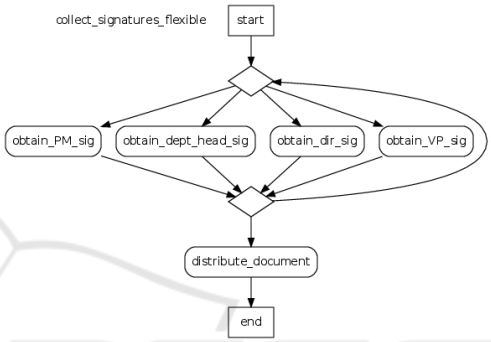

Figure 3: Flexible interpretation of Collect Signatures process.

A strict interpretation of this workflow requires that the signatures be obtained in order: Project Manager (PM) first, then Department Head, then Division Director, and finally Vice President of Engineering (see Fig. 2).

A flexible interpretation, on the other hand, recognizes that the document must have all the signatures before it can be distributed, but the order is not really important. As such, signatures could be obtained in any order, when the individuals are available. This flexible interpretation is depicted by the specification in Fig. 3. In this interpretation, the process iterates over signature collection, obtaining signatures one at a time, when the person is available (the diamonds indicate selection: "choose one of the following").

A final interpretation would be that each person could sign a copy of the signature page, and so the signatures could be collected in parallel. Once all signatures are obtained, the document can be sub-

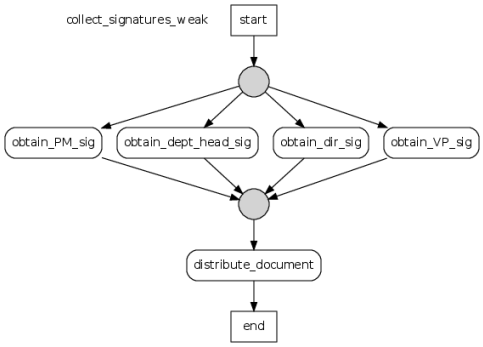

Figure 4: Weak interpretation of Collect Signatures process.

mitted. This is shown in Fig. 4 (the circles indicate a concurrent branch among all paths). Note that in this interpretation, the `requires` predicate for the `distribute_document` action enforces barrier synchronization: the `distribute_document` action cannot be performed until all of the signatures have been collected, which means all paths of the `branch` construct must complete.

Why not just use a specification that matches the weak (Fig. 4) interpretation? In some cases, this might be the best approach. However, there are some considerations that make the initial specification (Fig. 2) more appropriate. This specification captures the *intent* of the process: that the signatures act as a series of gateways to ensure that the Principle Investigator accepts responsibility for the document, and that the document meets with departmental, institute, and executive approval. A hierarchical approval sequence ensures that executives are not bothered with documents that don't meet departmental standards. At the same time, there are sometimes good reasons for circumventing this hierarchical sequence: for example, one or more individuals might be unavailable at the time their signature would be needed in the sequence, but could convey their intent to sign upon return; waiting for the strict sequence might result in missing the submission deadline. Also, our experience indicates that people tend to describe processes as sequential even when the sequence is not strictly needed; consequently, a sequential specification is initially easier to validate. Later, the specification might be evolved into a concurrent model as part of a process improvement exercise.

Our motivation for analysing context-sensitive flexibility comes from its relevance in modelling processes in regulated domains. The typical quality assurance standards that regulate safety critical domains, such as the ones that regulate medical devices or avionics software, have both requirements that need to be followed strictly, and parts which allow and in many occasions call for a context-sensitive flexibility of interpretation.

For illustration purposes, let us consider the Quality System (QS) Regulation – Medical Device Good Manufacturing Practices (Food and Drug Administration, United States Department of Health and Human Services, 2018a) from the U.S. Food and Drug Administration (FDA). Due to the fact that the QS regulation applies to a broad number of devices and processes, it follows a flexible approach which prescribes the essential elements to be incorporated in a manufacturer's quality process without prescribing specifically how to enact these elements. Furthermore, it is left to the manufacturer to determine which specific quality assurance procedures to implement according to their specific process or device.

This flexibility of enactment gives rise to a number of interesting questions such as:

1. A control flow perspective of process analysis: which enactment paths satisfy the regulatory requirements and which are rogue paths which could compromise compliance to the regulatory standards?
2. A data flow perspective of process analysis: can we highlight instances of resource black holes (where a resource is consumed by an action that produces no resources) and resource miracles (where resources appear to materialize out of nowhere, from actions that consume no resources)?

3 PML FORMAL SEMANTICS

We have defined not one, but three, distinct but related formal semantics for PML. This was done using a semantic framework known as the Unifying Theories of Programming (UTP) (Hoare and He, 1998) for two main reasons: first, the UTP framework makes it easy to formally relate the three semantic models. Also, it facilitates linking these semantic models to other application or domain specific resource semantics models.

3.1 UTP

UTP uses predicate calculus to define before-after relationships over appropriate collections of free observation variables. The before-variables are undashed, while after-variables have dashes. Some observations correspond to the values of program variables (v, v'), while others deal with important observations one might make of a running program, such as start (ok) and termination (ok'). The set of observation variables

associated with a theory or predicate is known as its *alphabet*. For example, the meaning of an assignment statement might be given as follows:

$$x := e \quad \widehat{=} \quad ok \implies ok' \wedge x' = e \wedge \nu' = \nu$$

Once started, the assignment terminates, with final x set equal to the e in the before-state, while the other variables (ν), remain unchanged. UTP supports specification languages as well as programming ones, and a general notion of refinement as universally-closed reverse implication:

$$S \sqsubseteq P \mathrel{\widehat{=}} [P \implies S]$$

Typically the predicates used in a UTP theory form a complete lattice under the refinement ordering, with the most liberal specification (*Chaos*) as its bottom element, and the infeasible program that satisfies any specification (*Miracle*) as its top. Iteration is then viewed as the usual least fixed point on this lattice.

3.2 Semantics by Three

A PML description can be viewed as a named collection of basic actions, defined in terms or the resources they need in order to start, and the new resources they produce once they have completed. These are all explicitly connected together using the control-flow constructs: `sequence`, `selection`, `iteration`, `branch`. However, this is also an implied flow-of-control ordering induced by the required and provides clauses of each basic action. Simply put, an action that requires resource r can't run until at least one other action runs that itself provides r. It is this tension between the explicit and implied control-flow that provides the basis for our three semantics for a PML description:

- **Weak:** control flow is completely ignored, and execution simply iterates the non-deterministic choice of actions whose resources are available (also known as the "dataflow interpretation").
- **Flexible:** the behaviour is guided by the control flow, but actions can run out of sequence if their required resources become available before the control flow has determined that they should start. In effect this allows re-ordering of sequences or execution of more than one selection, if resources allow.
- **Strict:** the behaviour follows strictly according to the control-flow structure, becoming deadlocked if control requests the execution of actions whose required resources are not available.

We can demonstrate that these three semantics form a refinement-chain, in the sense that any process enactment that satisfies the strict semantics also satisfies the flexible and weak semantics:

$$\text{Weak} \sqsubseteq \text{Flexible} \sqsubseteq \text{Strict}$$

Our collection of semantic models for any given PML description does not prescribe how such a description should be enacted. The choice of how to run a process depends entirely on the context in which it is being used.

3.3 Concurrency, Global State

Regardless of which of the three semantics we consider, a key common feature is that we are dealing with what is in effect a concurrent programming language with global shared state. Formal semantics for such languages are well established, in connected denotational and operational forms e.g. (Brookes, 1996), and more recently in UTP (Woodcock and Hughes, 2002). What all of these have in common, along with our three versions (Butterfield et al., 2016)(Butterfield, 2017), is that basically all such programs semantically reduce to a top-level iteration over a non-deterministic choice of all the currently enabled atomic state-change actions. In the case of PML, we consider the basic actions as atomic, being enabled if their `required` resources are present, and if applicable control-flow also permits the action to proceed. For the strict semantics, all the control-flow constructs are applicable, whereas for the weak view, none of them have any sway. In the flexible semantics, control flow constraints can be overridden by 'local' knowledge of actions not strictly scheduled, but actually able to start in terms of resource provision. By 'local' we usually mean actions contained in the `sequence` construct containing the actions in question.

3.4 Three Semantics Overview

The weak semantics is easiest: the alphabet consists of rs and rs' where rs is the set of resources present before an action or process runs, while rs' denotes the resources present once it has completed. A basic action first checks rs to see if its required resources are available. If they are not present, it will not run, otherwise it is available to be able to run. At any point in time, one of the basic actions that are so available is chosen to actually run, and it updates the resource set accordingly. So the meaning of

```
action A {
  requires { r1 }
  provides { r2 }
}
```

is the predicate

$$r_1 \in rs \wedge rs' = rs \cup \{r_2\}$$

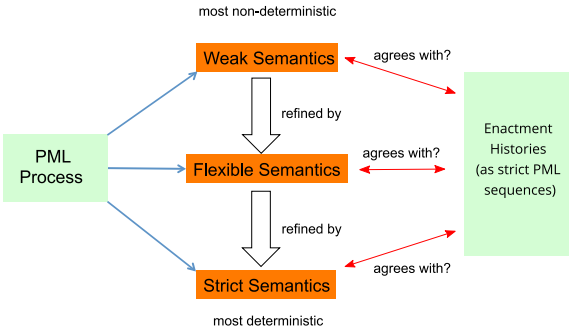

Figure 5: The three semantics for PML.

This predicate describes what must be true regarding rs and rs' in the event that this action actually ran. Its required resource r_1 must have been in the resource set before ($r_1 \in rs$), and its provided resource will have been added into the resource set afterwards ($rs' = rs \cup \{r_1\}$).

For the weak semantics, we simply collect up all the basic actions, discarding any information about the enclosing constructs, and then create a non-deterministic choice (logical-or) over all of them. This is enclosed in a loop that repeats until there is no change in the resources available. The weak semantics was first described in (Butterfield et al., 2016), along with a preliminary version of the strict semantics that wasn't fully compositional.

For the strict semantics, we keep the loop with choice structure used above for the weak semantics, but also add in extra alphabet variables in order to capture control-flow behaviour. This involves the generation of labels that identify when flow-of-control is about to enter or exit any statement (basic action or control-flow construct). We introduce a designated label-set (ls) that holds all the labels associated with the entry points of currently enabled statements. With every statement we associate an entry label (in) and an exit label (out). Typically the out label of one statement will correspond to the in label of the next statement to run. A basic action is now enabled when both its required resources are present, and its entry label is also in the label set (ls). When it has completed running, it will have updated rs as described above, but also ls, by removing its in label and adding its out label.

$$in \in ls \wedge r_1 \in rs \\ \wedge rs' = rs \cup \{r_2\} \wedge ls' = ls \setminus \{in\} \cup \{out\}$$

Careful use of a label generation mechanism and label substitutions allows us to give each statement a semantics independent of its context, but in such a way that it is easy for a more complex construct to use it and set it up in context. For example, the sequential composition of two instances of the above action would be achieved by generating a new label (mid, say), and substituting it in for the out label of the first instance, and the in label of the second. Then the two resulting predicates would be connected with logical-or. In effect, the variables in and out are used to manage contextual information in a compositional manner. Full details of the fully compositional strict semantics can be found in (Butterfield, 2017).

The flexible semantics requires a few more context-aware alphabet variables to propagate action enabling information to surrounding flow of control constructs, and is still under development.

3.4.1 Example

We can now consider how our three semantics deal with the Collect Signatures example (Figs. 1 to 4). Our semantics describes all the possible execution orders that can arise, given some starting state. With the strict semantics, applied to this example, we get only one sequence: `PM ; dept_head ; dir ; VP ; distr_doc` (here we shorten action name `obtain_XX_sig` to `XX`), which is precisely that of Fig. 2. With the weak semantics, the first four actions can occur in any order, while the fifth must wait until all the previous four are done. This gives 24 different interleavings of the four parallel actions in Fig. 4. In this case, because each arm of the selection in the flexible semantics has only one action (Fig. 3), we get the same 24 interleavings as for the weak case.

3.5 Current State

The current state of development of these semantic models is as follows — the extremal ones, weak and strict, are complete, while the flexible semantics has thrown up a number of interesting choices—there is a spectrum of possibilities here, depending on how 'local' the flexibility is. The current plan is to formalise the degree of flexibility that corresponds to the PML analysis and simulation tool described in (Atkinson et al., 2007).

4 RELATED WORK

Osterweil's 1987 paper (Osterweil, 1987) highlighted the importance of efficient software processes to deliver qualitative software and argued the idea of developing and using model processing languages that are similar to programming languages. Just a few years later, in 1993, a survey (Armenise et al., 1993) found that by then, there were quite a number of process modelling notations (the survey identifies Adele, ALF, APPL/A, DesignNet, Entity, EPOS, FunSoft, HFSP, Marvel, Merlin, MVP-L, Oikos and SPADE) that exhibited syntax similar to program languages.

To date, many different treatments of process flexibility have been proposed in the literature and recent extensive reviews of the field can be found in (Rosa et al., 2017) and (Cognini et al., 2018). Yet, many issues that come with flexibility remain to be solved, and of particular interest to us is the need for further research in the verification (ensuring correctness) of flexible business processes (Cognini et al., 2018).

To contextualize our approach we will refer to the taxonomy proposed by (Reichert and Weber, 2012) that identifies four types of process flexibility needs : (1) *Variability* – defined as the ability of providing different variants of the same process; (2) *Adaptation* – defined as the ability to (temporarily) deviate the execution path of a process; (3) *Looseness* – defined as the ability to execute a process when the decisions affecting the control flow are under-specified (and the execution path can be different in different runs); and, (4) *Evolution* – defined as a permanent modification of the process.

In this taxonomy our approach maps both into *Variability* and *Looseness*.

Additionally, of particular interest for us is the treatment of process flexibility seen from the lens of 'equal enough' processes by van der Aalst et al (van der Aalst et al., 2006), where a Petri nets approach is used to distinguish between negligibly different and completely different processes.

In our approach, we support the reasoning about flexible deployment by using the Unifying Theorems of Programming (UTP) (Hoare and He, 1998) framework for developing a range (which goes from "weak" to "strong") of formal semantics for PML.

5 CONCLUSIONS

In this paper, we discuss our insights into modelling context-sensitive (business and software) process flexibility. We developed formal semantics for a lightweight and flexibility-friendly process modeling language called PML. Our semantics cover three levels of interpretation for PML: strict, flexible and weak interpretations. We are using Unifying Theories of Programming (UTP) to model this range of semantic interpretations and the paper presents a high-level view of our formal approach. The added benefit of using the UTP framework is that we have semantically interoperable levels of interpretation for a process defined in PML. This can help to determine those situations where the interpretation level does not matter. Surprisingly this also gives us useful information about the nature of the process being modelled, and often guidance as to when flexibility is feasible or not. The formal semantics have applications in regulated domains, covering such areas as the certification process for medical device software. In particular, we can exploit the unification aspect of UTP to add in formal models of resources themselves to extend the scope of our analyses.

ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grants10/CE/I1855 and 13/RC/2094 to Lero - the Irish Software Research Centre(`www.lero.ie`).

REFERENCES

Aguilar-Saven, R. S. (2004). Business process modelling: Review and framework. *International Journal of production economics*, 90(2):129–149.

Armenise, P., Bandinelli, S., Ghezzi, C., and Morzenti, A. (1993). A survey and assessment of software process representation formalisms. *International Journal of Software Engineering and Knowledge Engineering*, 3(3):401–426.

Atkinson, D. C., Weeks, D. C., and Noll, J. (2007). Tool support for iterative software process modeling. *Information & Software Technology*, 49(5):493–514.

Brookes, S. D. (1996). Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163.

Butterfield, A. (2017). UTCP: compositional semantics for shared-variable concurrency. In da Costa Cavalheiro, S. A. and Fiadeiro, J. L., editors, *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*, volume 10623 of *Lecture Notes in Computer Science*, pages 253–270. Springer.

Butterfield, A., Mjeda, A., and Noll, J. (2016). UTP Semantics for Shared-State, Concurrent, Context-Sensitive Process Models. In Bonsangue, M. and Deng, Y., editors, *TASE2016*. under review.

Cognini, R., Corradini, F., Gnesi, S., Polini, A., and Re, B. (2018). Business process flexibility-a systematic literature review with a software systems perspective. *Information Systems Frontiers*, 20(2):343–371.

Food and Drug Administration, United States Department of Health and Human Services (2018a). Quality system (QS) regulation – medical device good manufacturing practices. https://tinyurl.com/y7uoefv8. viewed: 4th January 2019.

Food and Drug Administration, United States Department of Health and Human Services (2018b). Quality system regulation. https://tinyurl.com/y72mdqwr. viewed: 4th January 2019.

Hoare, C. A. R. and He, J. (1998). *Unifying Theories of Programming*. Prentice-Hall International, Englewood Cliffs, NJ.

Osterweil, L. (1987). Software Processes are Software too. In *Proceedings pf 9th International Conference on Software Engineering*, pages 2–13, Monterey, CA.

Reichert, M. and Weber, B. (2012). *Enabling flexibility in process-aware information systems: challenges, methods, technologies*. Springer Science & Business Media.

Rosa, M. L., Van Der Aalst, W. M., Dumas, M., and Milani, F. P. (2017). Business process variability modeling: A survey. *ACM Computing Surveys (CSUR)*, 50(1):2.

van der Aalst, W. M. P., de Medeiros, A. K. A., and Weijters, A. J. M. M. (2006). Process equivalence: Comparing two process models based on observed behavior. In *Proceedings of the 4th International Conference on Business Process Management*, BPM'06, pages 129–144, Berlin, Heidelberg. Springer-Verlag.

Woodcock, J. and Hughes, A. P. (2002). Unifying theories of parallel programming. In George, C. and Miao, H., editors, *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002 Shanghai, China, October 21-25, 2002, Proceedings*, volume 2495 of *Lecture Notes in Computer Science*, pages 24–37. Springer.