Representing Process Variation with a Process Family

Borislava I. Simidchieva, Lori A. Clarke, and Leon J. Osterweil

Laboratory for Advanced Software Engineering Research (LASER),
University of Massachusetts at Amherst, 140 Governors Drive, Amherst, MA 01003
{bis, clarke, ljo} @cs.umass.edu

Abstract. The formalization of process definitions has been an invaluable aid in many domains. However, noticeable variations in processes start to emerge as precise details are added to process definitions. While each such variation gives rise to a different process, these processes might more usefully be considered as variants of each other, rather than completely different processes. This paper proposes that it is beneficial to regard such an appropriately close set of process variants as a process family. The paper suggests a characterization of what might comprise a process family and introduces a formal approach to defining families based upon this characterization. To illustrate this approach, we describe a case study that demonstrates the different variations we observed in processes that define how dispute resolution is performed at the U.S. National Mediation Board. We demonstrate how our approach supports the definition of this set of process variants as a process family.

Keywords: process families, process variation, process variants, process instance generation, software product lines

1 Introduction

Process definitions are used as vehicles for improving coordination, communication, automation, and efficiency in teams that are developing software [19, 31]. Increasingly, process definitions are also being used to improve the functioning of teams in domains as diverse as manufacturing [10], medicine [7, 18], business [14, 40], and science [2, 33]. In all of these domains one can readily find processes that are widely used to discipline the way in which key aspects of the work are to be carried out. In earlier work we have begun to define the specific processes of interest in these domains; as we have elaborated these processes to lower levels of precise detail, however, we have started to observe that different team members often perform the process in ways that differ from each other. Although the differences may seem to be primarily differences in detail, process details can matter a great deal. Thus, it is necessary to consider how each of these differences produces a process variant, which is indeed a different process.

The existence of a proliferation of different processes would seem to complicate efforts to improve coordination, automation, and improvement in that it raises the question of which process is to be used to gain these improvements. Our observation is that some of the differences may indeed be profound, but that many differences might best be thought of as variations on a high-level process that is generally agreed

upon. If this is the case, then a large set of different variations might be thought of, and defined as, a process family, and that the process family might itself then be used as the basis for coordination, automation, improvement, training, etc.

In this paper we begin exploring the validity and applicability of the idea of process families. We attempt to characterize the sorts of variation that might be allowed within a family, and ways in which such variation might be represented. We validate our ideas by means of a case study, exploring the variation that we have observed in the course of defining the process of conducting a mediation at the U.S. National Mediation Board, and evaluating a specific approach to representing some of the principal forms of variation.

2 Related Work

Some particularly good summaries of work in software families, product lines, and vatiation are [30, 41]. Svahnberg et al. [38], in addition, present a taxonomy of different variability realization techniques. Jacobson et al. [20] describe some commonly used techniques to support software reuse, where variability is a main issue including inheritance, extension and extension points, parameterization, templates and macros, configuration and module interconnection languages, and generation of derived components. Other approaches include conditional compilation and dynamic binding [13], aspect- and feature-oriented programming [24, 35].

Generation approaches (e.g. [4, 8, 25]) seem particularly relevant to our work. In [8], the authors discuss the relation of feature models to various generative programming techniques such as inheritance and parameterization. In [34], the authors propose using component generators to support dynamically configurable components in software product lines. Moreover, these approaches often use a configuration specification as the basis for generation. This is similar to the notion of diversity interfaces introduced in the Koala model [39], where mechanisms such as switches, modules, and dynamic bindings of components capture variability and diversity. Jarzabek's XVCL (XML-based Variant Configuration Language) language [21] also describes systems in terms of variations and uses a generator to bind the variation points to specific variants. Work on decision models uses specifications to guide generation. KobrA [3] seems to have strong similarities to our proposed approach. KobrA extends UML models with decision models to describe the variability of components. In KobrA, each variation point is related to decisions and each component is associated with a decision model in addition to its structural, behavioral and functional models. A decision model is a list of decisions, a set of possible resolutions to each decision, and the possible effects of each resolution on the UML diagrams. The concept of decision models is also used in FAST [41] to support instantiation of domain models. Feature-oriented approaches have also been proposed to model variability. The FORM method [27] develops reusable and adaptable domain artifacts by using a feature model using AND/OR graphs where AND nodes indicate mandatory features and OR nodes indicate alternative features. A similar feature model is proposed by Griss et al [16]. Feature graphs, however, can become quite complex and unmanageable even for domains of reasonable sizes.

Our work is related to previous work in collaboration and group support systems, such as Group Systems ThinkTank or Facilitate.com, that support group decision making. Such tools implicitly define a process family by offering configuration options (e.g., are contributed ideas anonymous?) and by letting session "owners" change configurations dynamically (e.g., enabling categorization). We believe that our approach of providing vehicles for explicit representation of the process offers clear advantages. Groupware systems fall into three broad categories: 1) Systems that are "process-agnostic" such as Groove, WebEx, SameTime, or Caucus, 2) domain-specific tools, like the group support systems mentioned above, and 3) groupware toolkits (e.g, Lotus Notes) that support building groupware tools with a programmer-specified embedded process. Neither the "process-agnostic" nor the domain-specific tools support explicit representation of the process being executed, and while they may passively support a range of processes, they cannot support understanding the relationships between them. Similarly, while groupware construction tools support explicit coding and thus conformance of a single member to a process family, they cannot support clear representation of, and thus reasoning about, the family.

We strongly concur with Briggs, who argues that collaboration research should focus on "technology supported collaboration processes" instead of "collaboration technology" [6]. This idea is echoed in [26] that discusses the "emerging field of Collaboration Engineering" which is "an approach that designs, models, and deploys repeatable collaboration processes".

Much literature addresses modeling of software development processes, with an increasing focus on modeling workflows [14, 28, 40], business processes, and service architectures [1, 12]. There is far less focus on processes for government applications, and on negotiation and dispute resolution. Many approaches are based upon the use of a flowgraph model of the process [2, 29]. Others use such formalisms as finite state machines [17, 23] or Petri Nets [11, 15]. In some cases, the formalism recognizes the need to also model artifacts [37], often by using simple type systems. In fewer cases, the need to model resources and agents is also recognized, and again these models are usually simplistic [5, 9].

3 Approach

We propose applying the software product family approach to processes as a way to handle process variation. By creating a number of processes, which are all variations of one *metaprocess*, or alternatively, by creating one metaprocess to span an entire group of related processes, a process family is effectively generated. For several processes to be members of the same family, they must be sufficiently similar, i.e., contain a common core that is identical or slightly different across processes. Often, it may be difficult to identify a core but these processes may still belong to the same family if, at a higher level of abstraction, they are determined to be the same.

We hypothesize that all necessary process instances can be generated from a framework, perhaps with the aid of some sort of specification of required process goals. The fundamental approach to doing this is through composition of components that deal with three distinct principal process dimensions. This approach (illustrated in

Figure 1) is beneficial because it would allow for reuse of components across the process family and more importantly, the generation of new members of the family by simply compiling elements from common repositories.

We have experimented with the use of the Little-JIL process definition language [42, 43] as a vehicle for representing process families. Little-JIL is unusual in its clear separation of three concerns in process definition, namely the need for definition of 1) individual process steps and their coordination, 2) the behaviors of the agents that perform steps, and 3) the structure of the collection of artifacts that are produced and consumed by the steps. A complete Little-JIL process definition consists of one of each these three types of definition components. This is a particularly promising basis for modeling and defining process families, because each selection of a different set of these three components will generate a different process. In our work we have initialized the set and structure of steps with a fixed process, which we call the "Coordination Metaprocess," we then made varying augmentations with elaborative steps, while also making different selections of agent behaviors and artifact structures. This allowed for substantial process variation, and the totality of all such variants is what we call a process family.

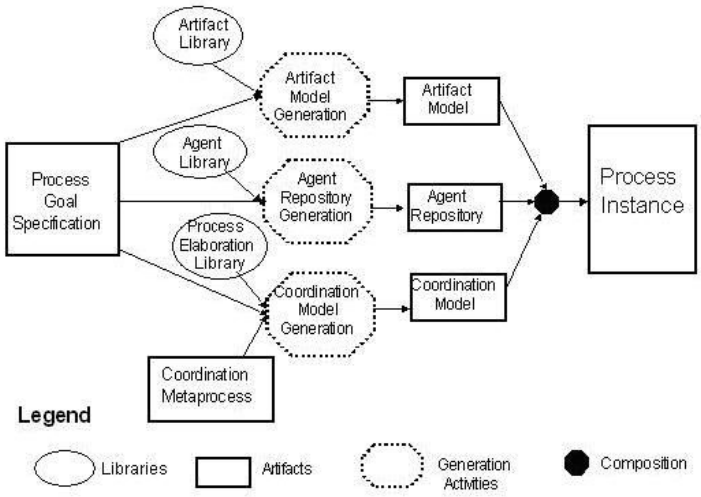

Fig. 1. Procedure for generating process instances.

As indicated in Figure 1, selected process elaboration instances could be drawn from a library to add elaborative details to the Coordination Metaprocess. The resulting coordination model instance is then composed with selected agent and artifact specifications. Selection of specific components from the appropriate libraries might be driven by the process goal specification. Understanding how this is done requires a short explanation of principal features of Little-JIL.

A Little-JIL process definition is a hierarchical decomposition of steps, each of which is executed by a specified agent. Steps communicate with each other by exchanging artifacts. Thus, a Little-JIL process definition consists principally of three

parts: 1) a coordination model that is a structure of steps represented by a "coordination diagram" such as shown in Figures 2 and 4, 2) a repository of agents, one of which is selected for late binding to execute a step (each step in the coordination diagram has an agent), and 3) a library of artifacts used, created, and transmitted by the steps of the process. Binding different agents and different artifact definitions to a given coordination model is thus a way to achieve process variation.

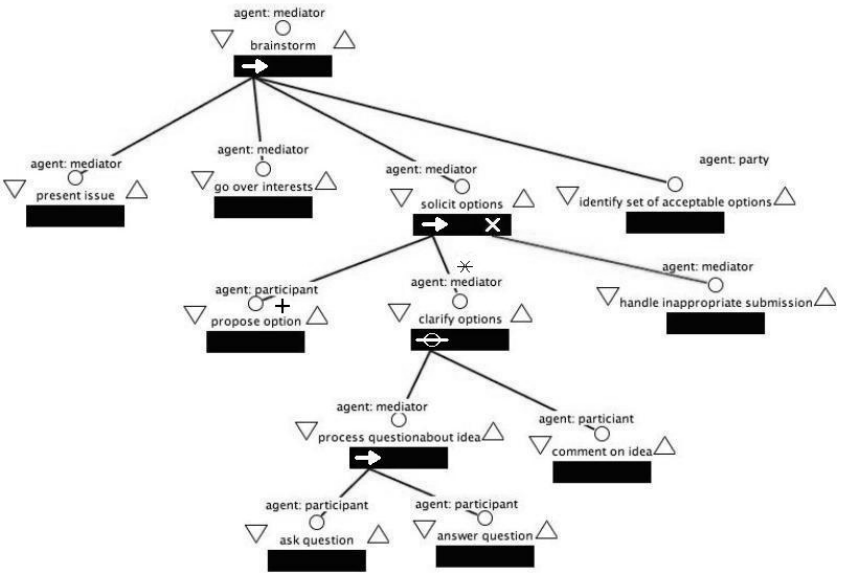

Fig. 2. The Little-JIL coordination diagram representing the "Brainstorm" process.

Figure 2 is a visual representation of a Little-JIL coordination model of a simplified brainstorming process. The coordination model is a hierarchy of process steps, each of which is depicted by a black rectangular "step bar". The step name is located above the step bar, which is accompanied by a set of badges that denote several semantic features. The behavior of a step is defined to be the behaviors of its children (the steps below the parent, connected to it by edges emanating from the left side of the step bar define the workflow, and the steps, connected to the parent by edges emanating from the right side of the bar are exception handlers). Leaf steps (i.e., those having no children) have no behaviors defined by the coordination model. Their behaviors are the behaviors of the step's assigned agent, enabling process variation in a way that will be illustrated.

Non-leaf steps contain a sequencing badge (imbedded in the left of the step bar), which defines the order in which its sub-steps execute. For example, a "Sequential" step (right-arrow in the "brainstorm" step in Figure 2) indicates that its sub-steps are executed in left-to-right order. In Figure 2, "solicit options" is a Sequential step, indicating that "propose option" and "clarify options" must be executed in this order.

Note that the step "propose option" is connected to its parent by an edge annotated

with "+" (a Kleene Plus). This indicates that "solicit options" has one or more copies of "propose idea" as its child(ren). The tag "agent: participant" specifies that any agent executing this step must be of type "participant". This obliges the agent repository to assign agents able to execute "participant" functions to each of these child steps. Thus the activity "solicit options" is defined to be one or more participants proposing options. Similarly, a * (Kleene Star) annotation means that the child step can be executed any number (zero or more) of times.

The "clarify options" step has a circle with a slash through it on the left of its step bar, which indicates that it is a "Choice" step. A Choice step is carried out by having its agent (in this case a mediator) make a choice between the various defined alternatives, each of which is defined to be a substep. Thus, in this case, the agent chooses between "process question about idea" and "comment on idea" as the way in which "clarify options" is to be executed. There are no other allowable alternatives.

Note that in this Coordination Model, the step "solicit options" consist of "propose option" and "clarify options." While the details of "propose option" are not specified (as it is a leaf step), the Coordination Model does specify that this step is capable of throwing an exception. This is inferable by the presence of an X on the right of the step bar of its parent, indicating that the parent, "solicit options," incorporates as part of its definition a subprocess that defines how to handle the throwing of an exception by any of its children. In this case, the X is connected to a child step, "handle inappropriate submission", presumably indicating that when the process of doing "propose option" results in the detection of offensive material, then the "handle inappropriate submission" step is invoked to delete that contribution. This exception handler is defined to be "complete," therefore the flow of execution continues as though the "solicit options" step has thereby been concluded.

Of additional importance is the way in which artifact flow is defined in Little-JIL. In the Coordination Model shown in Figure 2, the step "go over interests," for example, has a defined outgoing parameter, *interest list*, which is passed to its parent, the "brainstorm" step. Specifications of all the additional artifact flows in Figure 2 are elided here to reduce the complexity of the example.

Using Little-JIL, we can model process variation and generate instances of different processes within a family through several techniques. We have thus far identified three such process instance generating techniques:

Agent Behavior: by modifying the behavior of the agents executing the process (or parts of it), we create a variant of the process. Different selections from among different agent behaviors at execution time will create different process variants, but, moreover, additional variation is possible through the use of different semantics for defining the agent repository itself. For example, we propose to consider substituting for the previously described model in which objects have static collections of methods, a model in which the collection of methods used to define agent types is dynamic, responding to changes in process execution. This would take some control of the methods usable by an agent out of the agent's hands, thus creating the capability for additional process variation.

Task Elaboration: by "clipping on" small elaboration processes onto a leaf step, we can specify how to execute it differently. Since leaf steps contain no details specifying how they are to be executed, this decision is entirely up to the execution agent bound to the step. By adding elaborative substeps, the agent is mandated to execute the step

as defined by the behaviors of these children.
Artifact Structure: by selecting artifacts from a library of different artifact structures, featuring differences in semantics, structure, and content, we can create different process instances.

4 Case Study

In an ongoing research collaboration with the National Mediation Board (NMB), we have been developing a process definition for online dispute resolution (ODR) to be used by mediation professionals [22, 32]. We suggested that a rigorous process definition could be used to bring ODR technology into NMB, by indicating how computer technologies could be incorporated into these processes.

We initially believed that there was one single process to be defined and that this process had a well-defined goal, namely an agreement. But this project made it clear that our earlier belief was naïve and one process could not encompass all the variations introduced by the individual mediators. We found that not only do different mediators employ different processes from one another but an individual mediator might change the process, depending on the perceived effectiveness of the current process execution and changing group dynamics. These processes however, all seem to bear important familial relations to each other. Thus what the NMB context seems to call for is a process family, rather than a single process. As part of the case study, four descriptions of the "Brainstorm" process were elicited from four different mediation professionals. All four have attended the same training and must follow the same metaprocess prescribed by the NMB (partially outlined in Figure 2). We attempted to use the approach described above to see if all four elicited processes elicited could be best comfortably thought of as a process family.

5 Results

The "Brainstorm" coordination process shown in Figure 2 will now be taken as the Coordination Metaprocess that is to be used as the basis for a process family. We apply the three instance generation techniques outlined in section 3 to demonstrate how variation can be introduced to span a large set of variants, including all of the four different processes elicited from NMB personnel.
Agent Behavior Variation: recall that it seems necessary to support variation due to differences in the ways in which different agents perform an activity, perhaps under different circumstances during an ODR process execution. Thus, for example, different levels of anonymity might be desirable under different brainstorming circumstances, and this can be specified by defining differences in the ways in which different agents deal with the artifacts they must process. Figure 3 contains two different formal definitions of how an agent may deal with such artifacts. In this example, an agent is considered to be an instance of a type (in this case, the type is "Mediator"), and the definition of the type is in terms of the methods that it can

execute. We assume that the various methods are the various ODR process activities that the agent may be called upon to execute.

In this example, two different subtypes of agents of type Mediator are defined. Both definitions include a specification of how the step "Propose Options" is to be executed by detailing exactly how to execute the method "options list.add". The agent instance Mediator-1 of subtype Fully-Anonymous Mediator is defined so that it will never add to an options list any information about the contributor of a list item. On the other hand, the Mediator-2 agent instance of the Partially-anonymous Mediator subtype is defined to identify an options list item contributor only if the contributor is of type Mediator. Statically specifying either subtype will assure that the corresponding agent behavior is always executed, thus providing process variation. The possibility of execution time subtype binding affords the possibility of more variation in behavior. Clearly, one can populate an agent library with specifications of how each of the needed agent types is to execute each of the steps to which it might ever be bound in a full family of processes. In some cases, the agent specification might be null, in which case the agent would have no restrictions on its behavior.

```
Fully-Anonymous Mediator is-subtype-of Mediator:
Propose Options:
     for (Option opt: options) {
           options list.add(new Option(opt.what)); }
  Instances: Mediator-1{anonymous:yes}
Partially-anonymous Mediator is-subtype-of Mediator:
Propose Options:
     participant is-a mediator {
           for (Option opt: options) {
                 options list.add(new Option(opt.who,
opt.what)); }
Propose Options:
     for (Option opt: options) {
           options list.add(new Option(opt.what)); }
  Instances: Mediator-2{anonymous:partially}
```

Fig. 3. An example of items from the agent repository.

Or, as in the case shown in Figure 3, different agent subtypes might have different mandated behaviors, perhaps for the different steps to which they might be bound as agents or perhaps in response to different execution state details. The organization and structure of an agent repository is the subject of current research [36]. Thus, Figure 3 shows only one example of how this repository might be organized and defined. It is not clear that a type inheritance hierarchy will necessarily be used. It is conceivable that other agent definition approaches might be used at least to specify parts of agent behaviors. Subsequent research is expected to shed light on which agent definition formalisms, and which agent library organization strategies, seem most effective in supporting needed agent-based variation.

Task Elaboration: to demonstrate the task elaboration technique, we elaborate the "present issue" step that is a leaf in the Coordination Metaprocess in Figure 2. Until the process fragment of Figure 4 is bound to the "present issue" step of the process in

Figure 2, the way this step is to be executed is determined solely by the agent bound to it, subject to any restrictions or directions specified in the agent's definition.

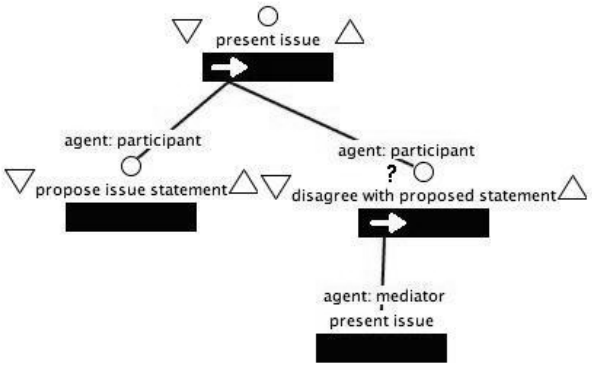

Fig. 4. "Present issue" process fragment elaboration

Once the process fragment of Figure 4 is bound, however, a new process instance (i.,e. a new member of the process family) is created. The new process instance differs from the previous process instance in that "present issue" now mandates that an issue statement is created and discussed by all participants iteratively until all agree to the statement (which means that the optional step "disagree with proposed statement" will not be executed, and "present issue" will not be called again), as directed by the process in Figure 4. Other process fragments could be defined to create other variations on this process. By late-binding different process fragments, new process instances are created and incorporated into the family. While late binding of step structures to leaf steps is not currently possible with the present version of the Little-JIL interpreter, this feature is to be included in future versions.

Artifact Structure Variation: as previously mentioned, the step "go over interests" in the Brainstorm process in Figure 2 has an outgoing parameter, *interest list*. The *interest list* is an unordered collection of the interests of the parties who are bargaining. Depending on which mediator professional is leading the mediation, he or she may choose to keep all interests together without duplicates, or alternatively, they can choose to keep each party's interests separate, without duplicates within the party.

This variation can be easily achieved by changing the artifact structure of *interest list*—by adding or removing an *author party* annotation to the structure of the *interest list* and by removing duplicates based on a predetermined comparison (e.g. if an *author party* annotation is present, two interests are the same only if both *contents* and *author party* are the same).

6 Discussion

Through applying the three outlined techniques for creating process variants through

process instance generation (changes in the agent behavior, artifact structure, or task elaboration fragments), we have been able to generate multiple instances that span most of the variations in the four processes elicited for the case study. By applying more than one technique simultaneously, much larger families can be generated.

This approach also provides an important vehicle for reuse in process definition by treating similar processes as a process family built from a common core (the Coordination Metatprocess) and a set of additional components, which augment the core. Moreover, by applying these techniques, it is easy to fine-tune large processes by switching components depending on the execution circumstances.

Although this approach to process families seems to be very promising, a considerable variety of additional work is also suggested. Initially, instance generation is to be done manually, based upon selection from libraries of agent repositories, artifact models, and process coordination elaboration models, respectively. Eventually, we expect that automation will at least facilitate, if not completely replace, manual selections from these libraries, although it is likely that human customization will always be needed to produce the three components to be composed into the final process instance.

Finally, it is important to gain a stronger sense what constitutes a process variant. As noted above, selecting different choices during execution seems quite different from creating different variants. Our case study did not provide us with much insight into how to distinguish between these two notions. It might be useful to address this problem by establishing formal metrics for determining the degree of similarity between processes, and perhaps use such metrics to guide decisions about what constitutes a viable variant.

Acknowledgements: The authors express their gratitude to Daniel Rainey of the National Mediation Board for providing the details of the ODR metaprocess. The authors also gratefully acknowledge Alexander Wise, Norm Sondheimer, Ethan Katsh, Leah Wing, Allan Gaitenby, M.S. Raunak, and Matt Marzilli for their support and insights about online dispute resolution, process formalisms, and Little-JIL.

This research was supported in part by the U.S. National Mediation Board and the US National Science Foundation under Award Nos. CCR-0204321 and CCR-0205575. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. National Mediation Board, the U.S. National Science Foundation, or the U.S. Government.

References

1. Alonso, G., Agrawal, D., Abbadi, A.E., Kamath, M., Gunthor, R., Mohan, C.: Advanced transaction model in workflow context. Proceedings of the 12th IEEE International Conference on Data Engineering, Proc. 12th International Conference on Data Engineering, New Orleans, February 1996 (1996) 574-581
2. Altintas, I., Berkeley, C., Jaeger, E., Jones, M., Ludäscher, B., Mock, S.: Kepler: An

Extensible System for Design and Execution of Scientific Workflows. Proceedings of the 16th International Conference on Scientific and Statistical Database Management, Santorini Island, Greece (2004) 423-424
3. Atkinson, C., Bayer, J., Muthig, D.: Component-based product line development: The KobrA approach. Proceedings of the The First International Software Product Line Conference, Denver, CO (2000) 289-309
4. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Transactions on Software Engineering and Methodology **1** (1992) 355-398
5. Belkhatir, N., Estublier, J., Walcelio, M.L.: ADELE-TEMPO: an environment to support process modelling and enaction. Software Process Modeling and Technology (1994) 187-222
6. Briggs, R.O.: On theory-driven design and deployment of collaboration systems. Int. J. Hum.-Comput. Stud. **64** (2006) 573--582
7. Clarke, L.A., Chen, Y., Avrunin, G.S., Chen, B., Cobleigh, R., Frederick, K., Henneman, E.A., Osterweil, L.J.: Process Programming to Support Medical Safety: A Case Study on Blood Transfusion. Proceedings of the Software Process Workshop 2005, Beijing, China. Springer-Verlag (2005) 347-359
8. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
9. Dami, S., Estublier, J., Amiour, M.: APEL: A Graphical Yet Executable Formalism for Process Modeling. Automated Software Engineering International Journal **5** (1998) 61-69
10. Deming, W.E.: Out of the crisis. MIT Press, Cambridge, MA (1982)
11. Emmerich, W., Gruhn, V.: FUNSOFT Nets: a Petri-Net based Software Process Modeling Language. IWSSD '91: Proceedings of the 6th International Workshop on Software Specification and Design, Como, Italy (1991) 175-184
12. Foster, H., Uchitel, S., Magee, J., Kramer, J., Hu, M.: Using a Rigorous Approach for Engineering Web Service Compositions: A Case Study. SCC '05: Proceedings of the 2005 IEEE International Conference on Services Computing (2005) 217-224
13. Gacek, C., Anastasopoules, M.: Implementing product line variabilities. Proceedings of the 2001 Symposium on Software reusability, Toronto, Ontario, Canada (2001) 109-117
14. Georgakopoulos, D., Hornick, M.F., Sheth, A.P.: An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases **3** (1995) 119-153
15. Ghezzi, C., Mandrioli, D., Morasca, S., Pezze, M.: A Unified High-Level Petri Net Formalism for Time-Critical Systems. IEEE Transactions of Software Engineering **17** (1991) 160-172
16. Griss, M., Favaro, J., d'Alessandro, M.: Integrating Feature Modeling with the RSEB. Proceedings of the 5th International Conference on Software Reuse (1998) 76-85
17. Harel, D., Naamad, A.: The {STATEMATE} semantics of statecharts. ACM Transactions on Software Engineering and Methodology **5** (1996) 293--333
18. Henneman, E.A., Cobleigh, R., Frederick, K., Katz-Basset, E., Avrunin, G.S., Clarke, L.A., Osterweil, J.L., Andrzejewski, C.J., Merrigan, K., Henneman, P.L.: Increasing Patient Safety and Efficiency in Transfusion Therapy Using Formal Process Definitions. University of Massachusetts, Amherst (2006)
19. Humphrey, W.S.: Managing the software process. Addison-Wesley, Boston, MA (1989)
20. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley Professional (1997)
21. Jarzabek, S., Zhang, H., Zhang, W.: XVCL: XML-Based Variant Configuration Language. Proceedings of the International Conference on Software Engineering, ICSE'03, Los Alamitos, CA. IEEE Computer Society Press (2003) 803-811
22. Katsh, E., Osterweil, L., Sondheimer, N.K.: Process Technology for Achieving

Government Online Dispute Resolution. Proceedings of the National Conference on Digital Government Research, Seattle, WA (2004)
23. Kellner, M.I.: Software Process Modeling Support for Management Planning and Control. Proceedings of the First International Conference on the Software Process, Redondo Beach, CA (1991) 8-28
24. Kiczales, G., Lamping, J., Mandhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming. Springer-Verlag (1997) 220-242
25. Knauber, S.J.a.P.: Synergy between Component-Based and Generative Approaches. Proceedings of ESEC/FSE-7, Toulouse, France (1999) 2-19
26. Kolfschoten, G.L., Briggs, R.O., Vreede, G.-J.d., Jacobs, P.H.M., Appelman, J.H.: A conceptual foundation of the thinkLet concept for Collaboration Engineering. International Journal of Human-Computer Studies **64** 611-621
27. Kyo, C., Kang, S.K., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain Specific Reference Architectures. 143-168
28. Leymann, F., Roller, D.: Workflow-Based Applications. IBM Systems Journal **36** 102-123
29. Mayer, R.J., al, e.: IDEF Family of Methods for Concurrent Engineering and Business Re-engineering Applications. Knowledge Based Systems, Inc. (1992)
30. Northrop, L.: Software Product Lines--Practices and Patterns. Addison-Wesley (2002)
31. Osterweil, L.J.: Software Processes Are Software, Too, Revisited. Proceedings of the 19th International Conference on Software Engineering, Boston, MA (1997) 540-558
32. Osterweil, L.J., Sondheimer, N.K., Clarke, L.A., Katsh, E., Rainey, D.: Using Process Definitions to Facilitate the Specifications of Requirements. Department of Computer Science, University of Massachusetts, Amherst, MA (2006)
33. Osterweil, L.J., Wise, A., Clarke, L.A., Ellison, A.M., Hadley, J.L., Boose, E., Foster, D.R.: Process Technology to Facilitate the Conduct of Science. Proceedings of the 2005 Software Process Workshop, Beijing, China. Springer-Verlag (2005) 403-415
34. Pavel, J.N.S., Royer, J.-C.: Dynamic Configuration of Software Product Lines in ArchJava. Proceedings of the Third International Software Product Line Conference (2004) 90-109
35. Prehofer, C.: Feature-Oriented Programming: a Fresh Look at Objects. ECOOP '07. Springer-Verlag (1997)
36. Raunak, M.S., Osterweil, L.J.: Effective Resource Allocation for Process Simulation: A Position Paper. Proceedings of the International Workshop on Software Process Simulation and Modeling, St. Louis, MO (2005)
37. Suzuki, M., Katayama, T.: Meta-Operations in the Process Model HFSP for the Dynamics and Flexibility of Software Processes. Proceedings of the First International Conference on the Software Process, Redondo Beach, CA. IEEE Computer Society Press (1991) 202-217
38. Svahnberg, M., Bosch, J.: A Taxonomy of Variability Realization Techniques. Software Practices and Experience **35** 705-754
39. van Ommering, R., Kramer, J., Magee, J.: The Koala Component Model for Consumer Electronics Software. IEEE Computer **33** 78-85
40. Weigert, O.: Business Process Modeling and Workflow Definition with UML (1998)
41. Weiss, D.M., Lai, C.T.R.: Software product-line engineering: a family-based software development process. Addison-Wesley (1999)
42. Wise, A.: Little-JIL 1.5 Language Report. Department of Computer Science, University of Massachusetts, Amherst, MA (2006)
43. Wise, A., Cass, A.G., Lerner, B.S., McCall, E.K., Osterweil, L.J., Sutton, S.M.: Using Little-JIL to Coordinate Agents in Software Engineering. Proceedings of the Automated Software Engineering Conference, Grenoble, France (2000)