Using Process Definitions to Support Reasoning about Satisfaction of Process Requirements

Leon J. Osterweil and Alexander Wise

Department of Computer Science
University of Massachusetts
Amherst MA 01003
{ljo, wise}@cs.umass.edu

Abstract. This paper demonstrates how a precise definition of a software development process can be used to determine whether the process definition satisfies certain of its requirements. The paper presents a definition of a Scrum process written in the Little-JIL process definition language. The definition's details facilitate understanding of this specific Scrum process (while also suggesting the possibility of many variants of the process). The paper also shows how these process details can support the use of analyzers to draw inferences that can then be compared to requirements specifications. Specifically the paper shows how finite state verification can be used to demonstrate that the process protects the team from requirements changes during a sprint, and how analysis of a fault tree derived from the Little-JIL Scrum definition can demonstrate the presence of a single point of failure in the process, suggesting that this particular Scrum process may fail to meet certain process robustness requirements. A new Scrum process variant is then presented and shown to be more robust in that it lacks the single of point failure.

1 Introduction

In earlier work the authors have suggested that software development processes seem to have much in common with application software [1, 2]. One similarity was that just as applications are defined using a programming language, so might the processes used to construct such applications also be defined using a *process* programming language. This has led to the development of several such languages, one of which (Little-JIL) is to be described later in this paper [3]. The focus of this paper, however, is on a different similarity, namely that both applications and the processes used to develop them should be designed so as to demonstrably satisfy their requirements. Just as applications must satisfy requirements (e.g. in dimensions such as functionality, speed, responsiveness, robustness, and evolvability), so must development processes also satisfy similar kinds of requirements. For example, software development processes must satisfy speed requirements, namely that they complete within the time limit imposed by customers and users. Development processes also have functional requirements that specify such things as the artifacts they are to produce (i.e. only code, code and design specifications, code and voluminous testing result reports, etc.).

This paper suggests the importance of devising a technology that supports the demonstration that software development process implementations meet specified process requirements, and suggests some initial capabilities for supporting that technology. At present the selection of development process approaches to meet process requirements is typically done informally, generally based upon intuition and anecdotal evidence. Thus, for example there is a general understanding that heavyweight processes such as those guided by the application of such approaches as the CMMI [4] can be expected to lead to software products that are relatively more robust, well- documented, and designed for evolvability over a long lifespan. On the other hand such processes seem to be less well-suited to the rapid production of software products, especially those that are not expected to be used for an extended period of time, or evolved in an orderly way. Indeed the popularity of agile approaches [5] seems to have derived largely from a sense that such development processes are more appropriate for the rapid development of software products, especially those that are relatively small, and destined for relatively short lifespans. While there is considerable anecdotal evidence to suggest that these intuitions are well-supported by experience, we suggest that the selection of a software development process should not be left entirely to intuition, and might better be done with at least the support of engineering approaches and technologies.

An essential support for this more disciplined approach to the selection of a software development process would seem to be technology for inferring the properties and characteristics of such processes. This, in turn, would seem to require that these processes be defined with sufficient detail to render them amenable to such analysis. Thus, this paper suggests how to use a process definition as the basis for inferring the properties of a process, so that the inferred properties might be compared to specifications of the requirements that the process is to fulfill.

The example presented in this paper is the use of analyses of a definition of the Scrum software development approach [6] written in the Little-JIL process definition language. Specifically, the paper introduces the Little-JIL language in section 2. In Section 3 Little-JIL is used to define a specific version of the Scrum software development approach. This definition itself is shown to immediately suggest some process properties and characteristics. Section 4 then demonstrates how analysis of the Scrum definition can derive a property that seems suitable for comparison to an expectable type of process requirement. The paper concludes with an evaluation of this process analysis technology in section 6, and a suggestion for future work in section 7.

2 The Little-JIL Process Definition Language

Little-JIL is a process definition language [3] that supports specification of processes involving different agent and non-agent resources. The language is defined through the use of finite state machines, which makes processes defined in Little-

JIL amenable to rigorous analysis. Its use of hierarchical decomposition supports the specification of arbitrarily precise process details.

The most immediately noticeable component of a Little-JIL process definition is the visual depiction of the coordination. The coordination specification looks initially somewhat like a task decomposition graph, in which processes are decomposed hierarchically into steps, in which the leaves of the tree represent the smallest specified units of work. The steps are connected to each other with edges that represent both hierarchical decomposition and artifact flow. Each step contains a specification of the type of agent needed in order to perform the task associated with that step. Thus, for example, in the context of a software development process, the agents would be entities such as programmers, testers, managers, the customer, etc. The collection of steps assigned to an agent defines the interface that the agent must satisfy to participate in the process. It is important to note that the coordination specification only includes a description of the external view and observable behavior of such agents. A specification of how the agents themselves perform their tasks (their internal behaviors) is a external to the Little-JIL process definition. Thus, Little-JIL enforces a sharp separation of concerns, separating the internal specification of what a resource is capable of doing and how the agent will do it, from the specification of how agents are to coordinate their work with each other in the context of carrying out the overall process.

While space does not permit a complete description of the language, further explanation of the notation will be provided in the context of the model of the Scrum software development method and interested readers are referred to [3] for a complete description of all of the features of the language.

3 Defining the Scrum Software Development Method

This section describes Scrum, a commonly used approach to the development of software, especially software that must be developed rapidly. Many papers, books, and courses have been developed to explain the Scrum approach, and there have been many additional papers and books written to describe experiences and anecdotes in the use of Scrum[6–10]. The fact that the descriptions of Scrum contained in these many diverse books, papers, and anecdotes are typically informal, and sometimes seem to be inconsistent with each other, has posed problems for those who wish to use Scrum, sometimes leading to uncertainty about whether the process actually being followed can or should be called "Scrum". This lack of rigor and agreement also made it difficult for us to decide just which description of Scrum should form the basis for our own work as described in this paper.

Ultimately, as the goal of this paper is to demonstrate the possibility, and the value, of inferring the properties of a process from the rigorous and precise definition of the process, the specific process definition chosen seemed to be less of an issue than the choosing of some specific process. On the other hand, the work described here does seem to also demonstrate the possibility that our ap-

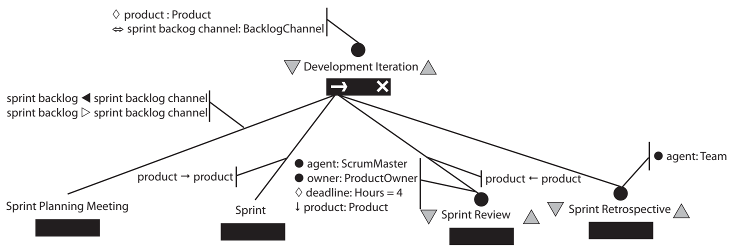

Fig. 1. The Scrum Development Iteration

proach of applying a process definition language can also be useful in identifying differences in opinion and interpretation of what is meant by a software development approach such as Scrum, and what the implications of these differences might be. The Little-JIL definition that is the basis for the work described here is based upon the informal Scrum description provided in [6].

As suggested by the preceding discussion, this process is defined hierarchically. At the highest level, Scrum involves three concurrent activities: the management of the product and release backlogs, and the iterative development of the product. For our example, we elaborate only the heart of the Scrum process: the iterative performance of "sprints." The *Development Iteration* step (Figure 1) specifies how one of these iterations is carried out. An iteration begins with a *Sprint Planning Meeting* to determine the work to be performed during the current iteration. This body of work is represented by the *sprint backlog* artifact and is stored in the *sprint backlog channel* to prevent concurrent updates. The Sprint Planning Meeting step is followed by the *Sprint* step in which the work is actually performed. The iteration concludes with the *Sprint Review* step and the *Sprint Retrospective* step. The Sprint Review step is a time-boxed meeting (note the diamond annotation that specifies a fixed deadline for completion of this step) led by the ScrumMaster agent with the support of the ProductOwner and team as resources. This step takes as an input artifact (note downward arrow annotation) the product artifact that was produced as the output of the Sprint step. The purpose of this Sprint Review step is to enable the team to discuss the results of the preceding sprint, and to close the loop between the product owner and the team. After the Sprint Review step concludes, the Sprint Retrospective is carried out with the team as the agent, indicating that the team meets in private to assess its performance during the last sprint as preparation for the next sprint.

3.1 Sprint

The *Sprint* subprocess (Figure 2) is the activity during which actual development work gets done. To be more precise, the Sprint process consists of 30 (note the 30 annotation on the edge connecting the Sprint parent step to its *Daily Sprint* child step) consecutive (note the right arrow step kind badge in the Sprint step)

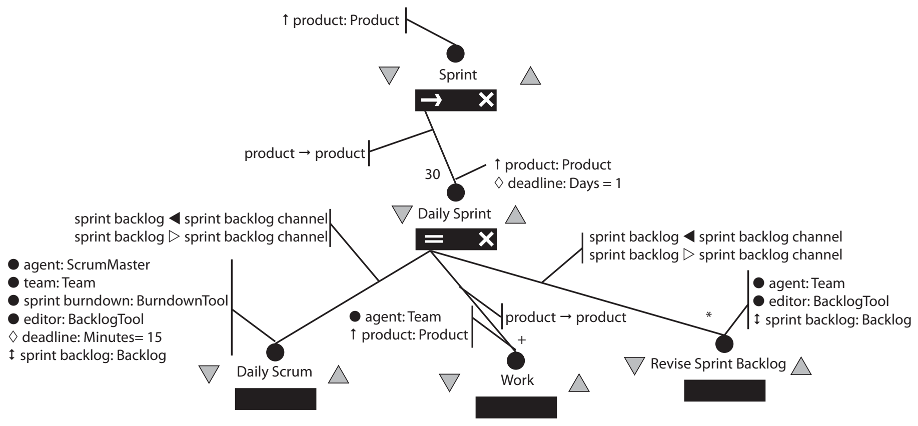

Fig. 2. Elaboration of the Sprint Step

performances of the Daily Sprint subprocess. As indicated by the = sign badge in the Daily Sprint step, this subprocess is carried out as the parallel performance of its three substeps, *Daily Scrum*, *Work*, and *Revise Sprint Backlog*. Both the Daily Scrum and the Revise Sprint Backlog steps require both access to, and update capability for, the sprint backlog. These accesses for these two steps are coordinated by using the sprint backlog channel to provide the needed concurrent access permissions.

The Daily Scrum step is a 15 minute (note the specification of this deadline by means of the diamond annotation) progress meeting during which the team meets to assess their progress. Note that the sprint backlog artifact is passed in as a parameter, and then also passed out of this step, after which it is written to the sprint backlog channel so that it is made available to the Revise Sprint Backlog step, which may be executing in parallel.

In addition to the execution of the Daily Scrum step, there are multiple performances of the Work step (note the + sign on the edge connecting the Work step to its parent). Each instance of the Work step produces a new version of the product artifact, presumably being comprised of more completed work items after the execution of this step. The agent for this step is the team.

Concurrently with the performances of the Work step there may be multiple performances of the Revise Sprint Backlog step (note the * on the edge connecting this step to its parent). The agent for this step is also team, and the effect of a performance of this step is to update the sprint backlog to reflect the addition or removal of tasks in the backlog.

4 Using Analysis to Determine Process Robustness

Any of a number of different process definitions could be derived by making different changes to the definition just presented. This paper makes no claim as to which of the resulting processes should still be considered to be Scrum processes. But this paper is intended to suggest that different analyses can be applied to these different processes to yield different characterizations. These different characterizations might then be useful in supporting decisions about which of the different processes seems appropriate for use in meeting different process requirements.

In this section we will demonstrate how analyses can be applied to the process model resulting in a characterization of the process that can be used to determine its adherence to two different process requirements. Additional kinds of analysis are suggested in section 7.

4.1 Applying Fault Tree Analysis

Fault tree analysis (FTA) is an analytic approach that is well known in safety engineering and other engineering disciplines, where it is used to identify the ways in which a specified hazard might arise during the performance of a process. More specifically, in this approach a graph structure, called a Fault Tree (FT), is built using AND and OR gates to indicate how the effect of the incorrect performance of a step can propagate and cause consequent incorrect performance of other steps. The analyst must specify a particular hazard that is of concern, where a hazard is defined to be a condition that creates the possibility of significant loss. Once such a hazard has been specified, FTA can then be used to identify which combinations of incorrect step performances could lead to the occurrence of the specified hazard. Of particular interest are situations in which the incorrect performance of only one step can lead to the creation of a hazard. Such a step, referred to as a single point of failure, creates a particularly worrisome vulnerability, and suggests that processes containing such steps are likely to fail to meet certain critical process robustness requirements. A complete treatment of the way in which an FT is generated from a Little-JIL definition is beyond the scope of this document, but the interested reader is referred to [11].

Identification of a Scrum Method Hazard One key Scrum process characteristic is that at the end of each sprint the ScrumMaster is always able to present a product that actually runs. Failing to be able to do this could be considered to be a hazard, and identifying a single step leading to such a hazard (a single point of failure) would suggest that such a process contains a robustness vulnerability. By applying FTA, Figure 3 shows that the previously presented Scrum process definition does not preclude the possibility of such a hazard resulting from a single point of failure as "Step 'Work' produces wrong 'product"' is sufficient for "Artifact 'product' from 'Sprint' [to be] wrong." Informally, we suggest that the previously defined process could be performed in a way that could be summarized as "write code for the first 29 days and then only on the 30th day make a first attempt to integrate everything." We hasten to note that this should not be taken as a weakness of the Scrum approach, because the Scrum process description upon which our definition is based explicitly states

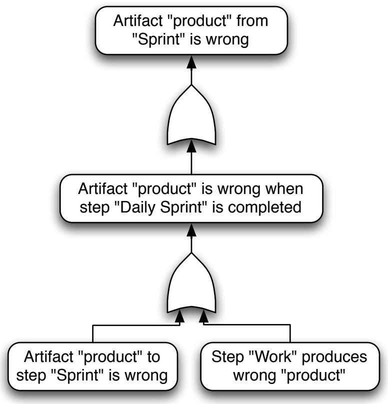
Artifact "product" from "Sprint" is wrong
Artifact "product" is wrong when step "Daily Sprint" is completed
Artifact "product" to step "Sprint" is wrong
Step "Work" produces wrong "product"

Fig. 3. Fault Tree Showing Single Point of Failure

that it is a management approach that does not incorporate any engineering practices. Any application of Scrum will of course incorporate such practices. What the analysis just presented demonstrates is that it is necessary to incorporate into the Scrum process definition appropriate product integration details that preclude the possibility of the indicated single point of failure.

Modeling Continuous Integration One way to build such details into the process definition is to integrate the Continuous Integration method into the Scrum method by replacing the step Work in the original Scrum definition, with a sub-process *Checked Work* (Figure 4). In Checked Work, the original Work step is followed by the *Integrate* step, whose purpose is specifically to integrate the work just completed with prior work products. The successful integration of this new work is verified by the performance of a post-requisite(represented by a upward-pointing triangle on the right of a step bar) to the Integrate step that verifies the correctness of the integrated artifact. If the verification does not succeed, then the *Rework* step is performed, to make whatever modifications are necessary in order to ensure that the required modifications are carried out. Details of the Continuous Integration method are not provided here, as the

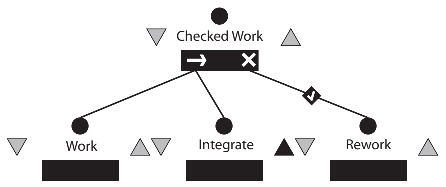

Fig. 4. Integration of Continuous Integration Into the Scrum Process Definition

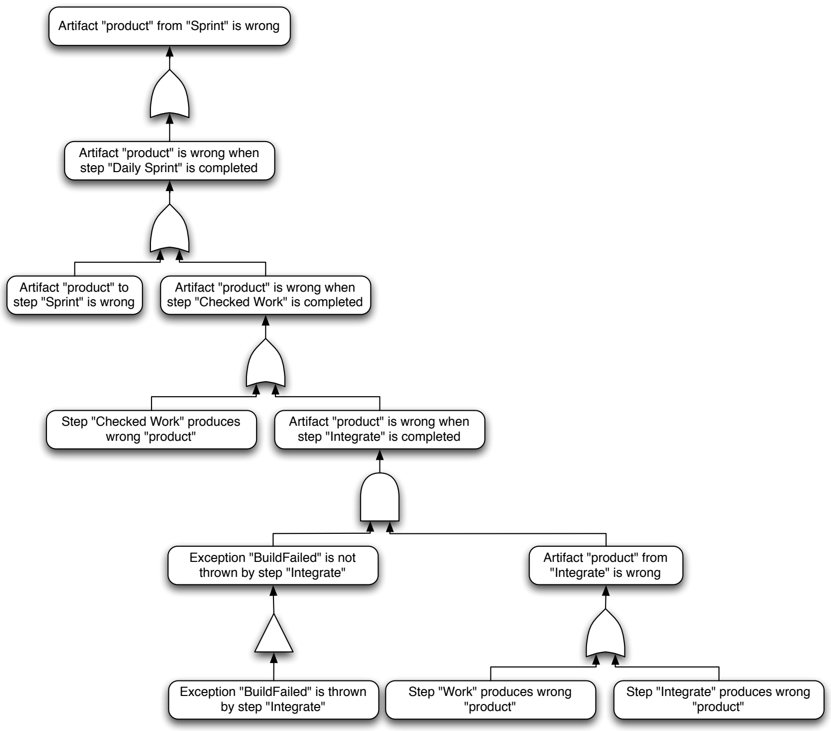
Artifact "product" from "Sprint" is wrong
Artifact "product" is wrong when step "Daily Sprint" is completed
Artifact "product" to step "Sprint" is wrong
Artifact "product" is wrong when step "Checked Work" is completed
Step "Checked Work" produces wrong "product"
Artifact "product" is wrong when step "Integrate" is completed
Exception "BuildFailed" is not thrown by step "Integrate"
Artifact "product" from "Integrate" is wrong
Exception "BuildFailed" is thrown by step "Integrate"
Step "Work" produces wrong "product"
Step "Integrate" produces wrong "product"

Fig. 5. Fault Tree Showing Fix

purpose of this section is to indicate how process rigor can support greater assurance about the success of method integration. Details of the Continuous Integration method definition would look very analogous to the details of the Scrum method definition.

As shown in Figure 5, the inclusion of Continuous Integration eliminates the single point of failure step in this definition. For this modified method, if ”Step ‘Work’ produces the wrong ‘product’ ”, then it will still be necessary for ”Exception ’BuildFailed’ is thrown by step ’integrate’ ” to fail to be thrown (note that the icon connecting ”Exception ’BuildFailed’ is thrown by step ’integrate’ ” to its parent is a complementation gate). Thus two steps must fail to be performed correctly in order for the hazard to occur, resulting in a process that meets robustness requirements that were not met by the previously-defined process.

4.2 Applying Finite-State Verification

Finite-state verification techniques are widely used to demonstrate the absence of specified event sequence defects from computer hardware and from software

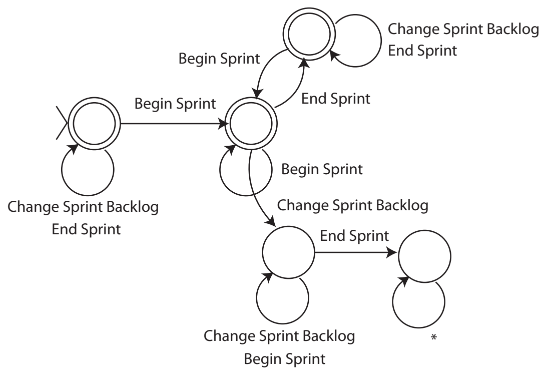

Fig. 6. Finite-State Machine Representation of the Property from PROPEL

code or designs. In [12], it is shown that Finite-state verification techniques may also be used to check if user defined properties hold in a Little-JIL process model. Finite-state verification is an analysis approach that compares a finite model of a system to a property specification. For example, we might specify an invariant that the development team is protected from requirements "churn" during a sprint, by asserting that the features being implemented cannot be changed except during the Sprint Planning Meeting (shown in Figure 1).

The FLAVERS[13] finite-state verification tool uses finite-state machines to specify properties that are usually created via PROPEL[14], a tool that allows the use of guided questions and disciplined natural language to aid the user in the creation of property specification. For our property, we have used PROPEL to specify that "Change Sprint Backlog" may not occur between a "Start Sprint" event and an "End Sprint" event, resulting in the finite-state machine shown in Figure 6.

Checking this property reveals that this property does not in fact hold for Scrum, producing an example trace, which after some preamble indicates that:

1. The step "Sprint" begins.
2. The step "Revise Sprint Backlog" occurs, changing the sprint backlog.

As this trace shows, the team is permitted to make changes to the sprint backlog during the sprint via the Revise Sprint Backlog step shown in Figure 2. Indeed, an anecdote in [6] describes a situation in which the Product Owner was bypassed and individuals were approaching team members and persuading them to make changes during the sprint. If we assume that the team members redirect any direct requests through the Product Owner as they are instructed to, then we can revise our analysis to disregard changes made to the sprint backlog via the "Revise Sprint Backlog" step, and then the property holds, showing that the Scrum process does indeed protect the team since only they themselves are able to make changes to the sprint backlog during the sprint.

5 Related Work

Many authors have addressed the problem of modeling, specifying, and defining various software development processes. Prime venues for this work include the International Conference on the Software Process (ICSP) series[15], and the journal Software Process Improvement and Practice (SPIP). Most of this literature attempts to model processes informally or with pictorial diagrams. In particular the Scrum approach to software development is described mostly in this way. Some examples are [6–10]. There have been a number of noteworthy attempts to create more rigorous notations with which to define processes (e.g. [16–19]). But these have typically not attempted to define popular software development processes. One notable exception is the Unified Process (UP) [20], which is defined precisely in terms of well-defined formalism. But even the UP definition is not then analyzed in order to infer properties for comparison with requirements. This paper seems unique in demonstrating that this is plausible and desirable.

6 Evaluation

While any clear graphical notation supports better understanding of a process, using a rigorously defined process language allows researchers and practitioners to evaluate, discuss, and improve the process based on analyses that are supported by a rigorous definition. The Little-JIL process definition language semantics are based on finite state machines to provide the precise semantic meanings of each of the language's features, which then serves as the basis for analyses that offer greater insights into such processes, support integration with other methods, and detect defects and vulnerabilities.

In this paper we showed how the Little-JIL Scrum method definition was used successfully for such analyses. First, a fault tree was generated automatically and used to identify a single point of failure in the process, suggesting a lack of process robustness that would presumably be worrisome, and facilitating the removal of that process vulnerability, as verified by analysis of a second automatically generated fault tree that demonstrated greater process robustness. Additionally, finite-state verification was used to show that the model does in fact demonstrate one of the key features of Scrum, namely that the development team is protected from requirements "churn" by the constraint that changes are routed through the Product Owner. We note that the finite- state verification tools and technologies used here have been shown to have low-order polynomial time bounds [13], which suggests that the work described in this paper should scale up from the small example presented here. Other work has demonstrated that these finite-state analyses can be applied to processes having hundreds of Little-JIL steps [21].

7 Future Work

This first demonstration of the use of process definition and analysis suggests that further kinds of analysis might also be successful in supporting additional demonstrations of the adherence of process definitions to additional desired requirements. For example, by attaching time limits to all steps (not just the steps in the example in section 3), and bounding the iterations (also necessary for finite-state verification) analyzers should be able to determine the maximum time required to carry out the specified process and thus determine if the process must always meet speed and responsiveness requirements. Discrete event simulation could also be used to study the adherence of processes to other kinds of requirements.

Finally we note that the application of these analyses to Scrum process definitions was used only as an example. These analysis approaches should be equally applicable to other types of processes to derive various sorts of properties and characteristics. Indeed this approach should be equally applicable to processes defined in other languages, as long as those languages have rigorously defined semantics. It seems particularly interesting to consider the possibility that some set of properties and characteristics might be identified that could be used to define the nature of such types of processes as Scrum. We suggest that a definition of what is quintessentially a "Scrum process" might well best be done by enunciating a set of properties and characteristics, rather than some canonical structure of steps. Analyses of the sorts suggested here might then be used to determine which actual processes are Scrum processes by subjecting them to analyses that determine whether or not the processes adhere to the defining set of properties and characteristics.

8 Acknowledgments

The authors gratefully acknowledge the many stimulating conversations with Aaron Cass, Bobby Simidchieva, M.S. Raunak, and Junchao Xiao, all of which have helped shape this paper. This work also builds fundamentally on the work of Bin Chen, Lori A. Clarke, and George S. Avrunin who have creating tools and technological approaches that enable the Finite State Verification and Fault Tree Analysis of process definitions. In addition Gordon Anderson's early versions of the Scrum process definition provided an invaluable starting point for this work. Financial support for this work was provided by the National Science Foundation under Award Nos. CCR-0427071 and CCR- 0205575, and a subcontract from Stevens Institute of Technology under the auspices of the DoD-sponsored Systems Engineering Research Center (SERC). 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 National Science Foundation, or the U.S. Government.

References

1. Osterweil, L.J.: Software processes are software too. In: 9th International Conference on Software Engineering (ICSE 1987), Monterey, CA (March 1987) 2—13

2. Osterweil, L.J.: Software processes are software too, revisited. In: 19th International Conference on Software Engineering (ICSE 1997), Boston, MA (May 1997) 540—548
3. Wise, A.: Little-JIL 1.5 language report. Technical Report UM-CS-2006-51, Department of Computer Science, University of Massachusetts, Amherst, MA (2006)
4. CMMI Product Team: CMMI for development, version 1.2. Technical Report CMU/SEI-2006-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburg, PA (August 2006)
5. Highsmith, J., Fowler, M.: The agile manifesto. Software Development Magazine **9**(8) (2001) 29–30
6. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice Hall, Upper Saddle River, New Jersey (2002)
7. Schwaber, K.: Agile Project Management with Scrum. Microsoft Press, Redmond, WA (2004)
8. Cohn, M.: Succeeding with Agile: Software Development Using Scrum. Pearson Education, Inc., Boston, MA (2010)
9. Scrum Alliance, Inc. http://www.scrumalliance.org/
10. Schwaber, K. http://www.controlchaos.com/
11. Chen, B., Avrunin, G.S., Clarke, L.A., Osterweil, L.J.: Automatic fault tree derivation from little-jil process definitions. In: 2006 Software Process Workshop (SPW 2006) and 2006 Process Simulation Workshop (PROSIM 2006). Volume 3966 of LNCS., Shanghai, China, Springer-Verlag (May 2006) 150—158
12. Chen, B., Avrunin, G.S., Henneman, E.A., Clarke, L.A., Osterweil, L.J., Henneman, P.L.: Analyzing medical processes. In: ACM SIGSOFT/IEEE 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany (May 2008) 623—632
13. Dwyer, M.B., Clarke, L.A., Cobleigh, J.M., Naumovich, G.: Flow analysis for verifying properties of concurrent software systems. ACM Transactions on Software Engineering and Methodology (TOSEM) **13**(4) (October 2004) 359—430
14. Cobleigh, R.L., Avrunin, G.S., Clarke, L.A.: User guidance for creating precise and accessible property specifications. In: ACM SIGSOFT 14th International Symposium on Foundations of Software Engineering (FSE14), Portland, OR (November 2006) 208—218
15. International Conference on Software Process. http://www.icsp-conferences.org/
16. Dami, S., Estublier, J., Amiour, M.: Apel: A graphical yet executable formalism for process modeling. Automated Software Engineering **5**(1) (1998)
17. Katayama, T.: A hierarchical and functional software process description and its enaction. In: Proceedings of the 11th international conference on Software engineering, Pittsburgh, PA (1989) 343—352
18. Kaiser, G., Barghouti, N., Sokolsky, M.: Experience with process modeling in the marvel software development environment kernel. In: 23rd Annual Hawaii Internationall Conference on System Sciences. (1990) 131—140
19. OMG: Software & systems process engineering meta-model specification. Technical Report formal/2008-04-01, Object Management Group (2008)
20. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley Longman, Inc., Reading, MA (1999)
21. Christov, S., Avrunin, G., Clarke, L.A., Osterweil, L.J., Henneman, E.: A benchmark for evaluating software engineering techniques for improving medical processes. In: International Conference on Software Engineering, Workshop on Software Engineering in Health Care (SEHC'10), Cape Town, South Africa (May 2010)