A Catalog of Software Development Rules for Agile Methods

Ulisses Telemaco[1], Renata Mesquita[1], Toacy Oliveira[1 2], Gláucia Melo[1], Paulo Alencar[2]

[1] COPPE – Systems Engineering and Computer Science Department
UFRJ – Federal University of Rio de Janeiro, Rio de Janeiro, Brazil
{utelemaco, rmesquita, toacy, gmelo}@cos.ufrj.br

[2] David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Ontario, Canada N2L 3G1
palencar@csg.uwaterloo.ca

Abstract. Background: Software Development is typically complex, unpredictable and dependent on knowledge workers. Software Processes are an attempt to handle development complexity by defining a set of elements such as workflows, roles, templates and rules. This paper focuses on software development rules which are restrictions applied to software process. Rules are often described in natural language and spread among many documents. Representing software development rules in natural language has at least two drawbacks: (a) Following rules become hard since it depends on the expertise of the workers and (b) Compliance checking - which is the act of verify whether rules are properly followed - is often performed without the support of specialized systems. The lack of a computational solution to guide the team or to support compliance checking is a problem since these activities performed manually are more error-prone, costly and less scalable if compared with automatic or semi-automatic approaches. **Problem**: Any automation initiative starts with the formal representation of compliance elements and a structured catalog that presents software development rules prone to formal verification is still missing. **Aim**: This study aims at identifying a set of software development rules based on agile methods and that may be susceptible to a formal verification. **Method**: A literature review as well as semi-structured interviews with practitioners were conducted to identify a set of agile software development rules. **Result:** We identified a set of 10 rules for agile methods that were considered relevant according to research criteria. The rules are presented in a structured catalog.

Keywords: software process rules; automatic process compliance checking; catalog software process rules; agile method rules

1 Introduction

Software industry has faced many challenges: tight deadlines, short time-to-market, low budgets (do more with less), necessity to offer high ROIs (return on investments), technologies always evolving and changing, high competition, just to name a few. In

such environment, it is fundamental for software companies are able to develop software in a flexible and adaptive manner. In response to this demand, Software Processes have become a complex mission involving the collaboration of several workers who perform non-trivial tasks orchestrated by a hard-to-predict model [1]. Some Software Processes share many characteristics with Knowledge Intensive Process [2], including: *Knowledge-driven*, *Collaboration-oriented*, *Unpredictable*, *Emergent*, *Goal-oriented*, *Event-driven*, *Non-repeatable* and *Rule-driven*. *Rule-driven* means that developers may be influenced by or may have to comply with constraints and rules that drive actions and decision making. In some areas, such as medical and financial sectors, it is necessary - and sometimes legally required - to certify that a software was developed according to established guidelines, norms and government legislation.

In fact, a worker has to handle a wide range of software development rules that has different origins such as: (a) technologies constraints which impose, for example, the correct order that some tasks should be performed; (b) process restrictions defining role permissions, how iterations and teams should be organized, which templates should be used, etc; (c) company policies stating that a worker is not allowed to work more than 8 hours per day or teams should not work on holidays; (d) legal regulation that states that every feature in the software should have an unit test; (e) project management practices defining desired profit margin, maximum number of developers in a team etc. Typically these rules are described in natural language and spread among many documents.

The use of natural language to describe software development rules and consequently the lack formal representation have some weaknesses: (a) Since software development rules are mainly described in textual documents, they become implicit in the process. Projects become dependent on the experience and knowledge of workers in order to comply with these rules. The problem is worse for newbies who have to work without having mastered those implicit rules. (b) Compliance checking for software projects is often conducted by Q&A consultants who have to manually verify whether the rules have been followed. The lack of automation in compliance checking can turn the activity more error-prone, costly and less scalable [3], [4].

The study presented in this paper is part of a comprehensive research that is pursuing formal representation and verification of software development rules. To the best of our knowledge, a catalog that presents software development rules in a structured format is still missing. Thus, the goal of the paper is to fulfill this gap by identifying a set of software development rules present in agile methods that are susceptible to formal verification. We are aware that software development rules are not exclusive from agile development, nevertheless we decided to focus on agile methods as a strategy to reduce the scope of observation.

We conducted a literature review that allowed us to extract 20 rules that were organized in a preliminary catalog. This catalog was subject of a survey with specialists and practitioners that aim at verifying the relevance of identified rules. The analysis of these experiments resulted in a structured catalog containing 20 software development rules for agile methods. For sake of space, we presented a detailed description of the 10 most relevant rules (according to criteria defined in the study).

The remainder of the paper is organized as follow: section 2 presents the background and section 3 discusses the research methodology. The catalog of software development rules for agile methods is presented in section 4. Discussions are presented in section 5. Section 6 discusses the Related Work and section 7 concludes the paper and presents future work.

2 Background

2.1 Software Development Process

In the context of the software industry, a Software Process (SP) can be represented by the set of activities, artifacts, roles and restrictions to develop or maintain a software system [5]. Feiler [6] define a SP as the set of partially ordered steps intended product or enhance a software product. According to Munch et al [7] SP is a goal-oriented activity in the context of engineering-style software development.

In order to become more efficient and competitive, organizations have incorporated Software Engineering concepts and best practices into their SPs[8]. As a result, SPs become highly flexible, non-repeatable, less formal, depend on the developer knowledge and accept some level of improvisation and creativity. According to [9] SPs typically have characteristics such as: (a) They depend intensely on people and context; (b) They are unpredictable and emergent; (c) Many activities do not support automation; (d) They depend on communication, co-ordination and cooperation between people; (e) Running an instance can take a long time and (f) Process may change during its execution.

Therefore, developing a software system is not a trivial process. According to the study presented by DiCicio [2] - which discussed many aspects of the so-called 'Knowledge Intensive Processes' and introduced a spectrum to classify a Business Process according to its characteristics - a SPD could be classified as Structured with ad hoc exceptions or Unstructured with-predefined segments). An important characteristic of Knowledge Intensive Processes (which include Software Processes) is that they are Rule-Driven [2]. The process is highly influenced by a set of software development rules.

2.2 Agile Methods

In 2001 a group of practitioners and researchers proposed the so-called Agile Manifesto [10], [11]. The manifesto described core values and principles for software development in response to the appeal of an eager community asking for lighter, faster and nimbler software development processes. Several agile methods were proposed inspired by the Agile Manifesto including XP [12], Scrum [13], Crystal methods [11], ADS [14], DSDM [15], FDD [16], OpenUp [17] and others.

The boundaries that divide agile and traditional (or non-agile) methods (if they exist) are not clear. Abrahamsson et al [18] tried to define "What makes a development method an agile one?" and concluded that a software development is 'agile' if it is: (a)

incremental (small software releases, with rapid cycles); (b) cooperative (customer and developer team working together with close communication); straightforward (the method itself is easy o learn and to modify, well documented); (d) adaptive (able to make last moment changes).

2.3 Software Development Rules

In the context of this paper, a Software Development Rule (Rule for short) represents a restriction or desire that is applied to a Software Project and is prone to formal verification. A Software Development Rule could have many origins such as Software Process, Technology, Organizational Policies, Contractor Restrictions, Legislation just to name a few.

It is important not to confuse Software Development Rules with Software Business Rule, which are rules related to the Software produced. Software Business Rules are not included in the scope of this study.

3 Study Methodology

In the next sections we discuss the literature review and the survey conducted in this study.

3.1 Literature Review

A literature review was the first step in our study that aims at compile a catalog of rules for agile development. The rest of this section presents details about the literature review.

Aim: The aim of the literature review was to identify, in the specialized literature, agile development practices, suggestions, constraints or restrictions that are prone to formal verification.

Methods Selection: The agile methods were selected according to the criteria presented by Abrahamsson [18] (and summarized in section 2.2). As a result, the following method were included in this study: Extreme Programming [19][20][12], Scrum [21][22][13][23], Crystal family of methodologies [11], Feature Driven Development [24][16], Dynamic Systems Development Method [15], Adaptive Software Development [14] and OpenUp [17].

Paper Selection: We selected primary studies and grey literature such as agile method documentation.

Extraction Strategy: The documents were fully read in order to find rules (practices, suggestions, constraints or restrictions prone to formal verification). The assessment of this criterion was made based on the personal experience of the authors. In order to minimize potential threats to the validity of this strategy, we submitted the selected rules to a survey with practitioners. The survey is described in the next section.

Results: We extracted initially 46 software development rules. After a detailed analysis, 10 duplicate rules and 16 less relevant or less prone to formal verification were

removed from the initial set. The result of the literature review was a structured catalog with the 20 rules (showed in Table 1 in alphabetical order):

1. *Avoid Complex Tasks*	11. *Every Iteration should be timeboxed*
2. *Avoid Dependence on Internal Specialists*	12. *Every Iteration should have a deliverable*
3. *Avoid Shared Developers*	13. *Every Iteration should have an Iteration Planning*
4. *Avoid Unplanned Work*	14. *Every Iteration should have an Iteration Retrospective*
5. *Concurrent Iterations are not allowed*	15. *Every Iteration should have an Iteration Review*
6. *Deliveries should be frequent*	16. *Every It. Tasks must be estimated before starting Iterat.*
7. *Development should be Test-driven*	17. *Every meeting should be timeboxed*
8. *Development Team should be small*	18. *Goals should be clear and well-defined*
9. *Do not finish an Iteration with open Tasks*	19. *Higher priority tasks must be executed first*
10. *Every Iteration should be the same length*	20. *There should be no break between Iterations*

Table 1. Initial catalog of Software Development Rules for Agile Methods.

3.2 Survey

In order to confirm and complement results obtained with the literature review, we conducted a survey based on semi-structured interviews [25] with practitioners to find the relevance of the identified rules under the industry perspective. The rest of this section presents the study design that was elaborated based on the protocol presented by Oishi [26].

Survey Planning.
Aim: The aim of the survey was to evaluate the relevance of the rules identified in the literature review.
Research Question: The survey aimed at responding the following questions: (a) Which rules (present in the given catalog) are relevant and prone to formal validation; (b) Which rules (present in the given catalog) are not relevant or not prone to formal validation; (c) Which evaluation strategies (present in the catalog) are coherent with industry practice? (d) Which evaluation strategies (present in the catalog) are not coherent with industry practice and therefore should be removed from catalog?
Instrumentation Details: The material used in the study included an online questionnaire developed using Google Sheets and divided in three sections: (a) Subject Characterization, (b) Organization Characterization and (c) Agile Rules.

The Agile Rules section contained the list of 20 rules collected from literature review. The participants should indicate, for each rule, the relevance of the rule according to their personal/professional opinion. They also should indicate how relevant could be the formal verification of the rule. The questionnaire accepts the following answers: *No relevance*, *Slightly relevant*, *Very relevant* or *Absolutely relevant*.
Selection of Subjects: We applied a convenience sampling [27] and participants were selected from our professional and academic networking connections. The criteria for

the selection of participants were: (a) the participant should have experience (at least 5 years) as Project Manager or Quality Assurance Consultant and (b) the participant should work (or have worked) in an organization that adopts a software process based on agile methods. We avoid selecting participants that are aware of this research (so we excluded coauthors and coworkers).

Survey Execution.
After planning the study, we conducted a preliminary execution using subjects from inside our research group. The data from this execution was not considered in the final results. Our goal was to collect feedback from the participants and assess the interview plan.

Initially 10 candidate subjects were chosen to be interviewed. We focused on practitioners working on agile projects with relevant experience in this topic. We received nine positive replies. For eight interviews, the first part (study contextualization) was conducted online and for one subject personally. The selected subjects included 7 Software Project Managers and 2 Quality Assurance Consultants.

4 Catalog of Software Development Rules

In this section we present the catalog of software development rules and discuss briefly how it was elaborated. We identified from a literature review an initial catalog with 20 rules that was submitted to a survey in order to assess the relevance of the rules. For the sake of space, we have selected the 10 most relevant rules (according to criteria presented below) to be detailed in the paper.

To rank the rules and attribute a degree of relevance, we used as criteria: (a) number of agile methods that cited the rule and (b) relevance according to the opinion of the survey participants.

The 10 rules that will be detailed in the paper are (showed in relevance order):

1. Higher priority tasks must be executed first	*6. Every meeting should be timeboxed*
2. Deliveries should be frequent	*7. Every Iter. should have an Iter. Planning*
3. Every Iteration should have a deliverable	*8. Avoid Complex Tasks*
4. Every Iteration should have an Iteration Retrospective	*9. Avoid Unplanned Work*
5. Goals should be clear and well-defined	*10. Avoid Dependence on Internal Specialists*

Table 2. Initial catalog of Software Development Rules for Agile Methods

4.1 Rule 01 - Higher priority Tasks must be executed first

The rule states that the team must work on the highest priority tasks first.

Agile Methods:	*Higher priority Tasks must be executed first* is mentioned by four methods: Scrum, Crystal Methods, DSDM and OpenUP. For Scrum, the whole team should focus on Sprint goal. In Crystal methods, the project leader should prioritize the goals which allow developers to focus on particular areas. In DSDM, to fulfill the principles *Focus on the Business Need*

	and *Deliver On Time*, DSDM teams must focus on business priorities. OpenUP teams self-organize around how to accomplish iteration objectives and commit to delivering the results.
Industry Perspective:	All survey participants confirmed that working in higher priority tasks is an important rule. However, a participant mentioned that exceptions are tolerated in situations where it is not possible to work on high priority tasks. As example, he cited a situation where an available worker does not have the required skills to perform a high-priority task. In this case, the worker is allowed to work on less important tasks.
Evaluation Strategy:	The rule could be verified assessing the execution history.
Parameters:	No parameter was identified for the rule.

4.2 Rule 02 - Deliveries should be frequent

This rule states that a Software Project should have frequent deliveries. The concept of continuous deliveries is very important to agile methods and this rule is almost a mantra between agilists.

Agile Methods:	Deliveries should be frequent is mentioned by all methods analyzed. XP divides the work in Small and Short Releases. In Scrum the work is broken in Sprints which are timeboxed periods (approximately 30 days) where the team produces a new executable version of the software. In Crystal methods the development is also incremental and the release times depend on the length of the project. While in Crystal Clear delivery internals are periods of two to three months, in Crystal Orange the increments can be extended to four months. In FDD iterations should take from a few days to a maximum of two weeks. DSDM's philosophy states "best business value emerges when projects are aligned to clear business goals, deliver frequently and involve the collaboration of motivated and empowered people". In ADS the project is broken in units called Adaptive Development Cycles that typically last between four and eight weeks. OpenUP divides the project into iterations that take a few weeks.
Industry Perspective:	All participants indicated - with different levels - that the rule is relevant. No additional comment was given.
Evaluation Strategy:	An approach to evaluate the rule is to assess the interval between two iterations with deliverable.
Parameters:	A *Desirable Delivery Interval* parameter could be used to customize the rule.

4.3 Rule 03 - Every Iteration should have a deliverable

The concept of continuous delivery is fundamentally important for agile methods that argue that the team should deliver a new version of the software at the end of each iteration.

Agile Methods:	The rule is mentioned by five agile methods: Scrum, FDD, DSDM, ADS and OpenUp. For Scrum it is desired that the team deliver a new version of the software at the end of each Sprint. In FDD at least one new feature should be delivered at the end of an Iteration. A key factor for the success of principle *Deliver on Time* in DSDM is that at the end of each iteration, the team show a deliverable. In ADS at least one new component should be delivered at the end of a Development Cycle. The practice *Iterative Development* in OpenUp defines that an iteration should not be extended without any software to be demonstrated.
Industry Perspective:	All participants indicated - with different levels - that the rule is relevant. No additional comment was given.

Evaluation Strategy:	An approach to evaluate whether every iteration has a deliverable is to use a specific field to describe the deliverable of an iteration.
Parameters:	A *Tolerated Number of Consecutive Iterations Without Deliverable* parameter could be used to make the rule more flexible.

4.4 Rule 04 - Every Iteration should have an Iteration Retrospective

Retrospectives represent opportunities for the team to reflect on how they are working and improve the method when necessary. The rule states that a retrospective ceremony should be held at every iteration.

Agile Methods:	Retrospective ceremonies are mentioned by three agile methods: Scrum, Crystal methods and ADS. Scrum defines Sprint Retrospect as a meeting that usually follows the Sprint Review, where the team (and only it) gives and receives feedback on the process followed during the Sprint. Crystal encourages a practice called *Reflective Improvement* in which developers take a break from regular development and try to improve their processes. The principle *Learning Loop* in ADS is also based on Project Retrospectives that are usually carried after each cycle.
Industry Perspective:	All participants indicated - with different levels - that the rule is relevant. No additional comment was given.
Evaluation Strategy:	We identify two alternatives to verify the presence of a retrospective ceremony. One alternative is to check if there is any task in the iteration plan that represents the Retrospective meeting. Another alternative is to verify if there is any task in the iteration plan that produces an Iteration Retrospective Summary artifact.
Parameters:	A *Tolerated Number of Consecutive Iterations Without Retrospective* parameter could be used to make the rule more flexible

4.5 Rule 05 - Goals should be clear and well-defined

Project and Iteration Goals should be clear and well-defined.

Agile Methods:	*Goals should be clear and well-defined* is mentioned by methods Scrum, Crystal, DSDM, ADS and OpenUp. Scrum states that Sprints should have well defined goals. In Crystal methods, goals should be clear and developers should know exactly what the goals of the project are. The principle *Focus On The Business Need* in DSDM defines that every decision taken during a project must be guided by project goals. Thus, it is important that these goals are well defined and communicated to all team. In ADS the development is Mission-oriented (or goal-oriented) and the activities in each cycle must be aligned with the project mission. Thus, having a well-defined and clear goal is a key factor for ADS method. OpenUP teams self-organize around how to accomplish iteration objectives.
Industry Perspective:	All participants indicated - with different levels - that the rule is relevant. No additional comment was given.
Evaluation Strategy:	An approach to evaluate *Goals should be clear and well-defined* rule is to use a specific field to describe iteration goals.
Parameters:	No parameter was identified for the rule

4.6 Rule 06 - Every Iteration should be timeboxed

The rule defines that iterations should have a fixed time duration. Thus, an iteration should not be extended or shortened to fit planned or unplanned features.

Agile Methods:	*Every Iteration should be timeboxed* is mentioned by four methods: Scrum, DSDM, ADS and OpenUp. Scrum method defines that a Sprint should be timeboxed. In DSDM, the principle *Deliver On Time* states that delivering a solution on time is a very desirable outcome for a project and is quite often the single most important success factor. In order to achieve this principle DSDM teams should need to, among other things, timebox work. ADS method argues that ambiguity in complex software development can be alleviated by fixing tangible deadlines on a regular basis. The practice *Iterative Development* in OpenUp defines that do not extend an iteration in order to finish work.
Industry Perspective:	Regarding the survey, there was no unanimity among the participants. Some participants said timeboxing should be rigid followed. Other subjects said that timeboxing is desired, but not a rigid rule, and changes in iteration duration are a common practice. For these participants, is preferable to extend an iteration in order to include important features than achieve timeboxing with less features.
Evaluation Strategies:	Timeboxing could be verified assessing if there was any variation in the iteration duration after it has been planned.
Parameters:	A *Tolerance Of Change* parameter (absolute or percentage value) could be used to indicate the maximum variation tolerated. For example, a 5% tolerance means that an iteration could have their duration shortened or extended by 5% of its baseline duration.

4.7 Rule 07 - Every Iteration should have an Iteration Planning

This rule states that every iteration should be planned before its start. Normally an iteration plan is elaborated with the main stakeholders (developer team and customer) that together decide what should be developed in the iteration. Long term plans are made only in a high level of detail.

Agile Methods:	*Every Iteration should have an Iteration Planning* is mentioned by all agile methods investigated. The practice *Planning Game* in XP promotes a close interaction between team and customer. Developer estimate the effort for implementing customer stories and customer then decides about timing and scope of the deliverables. Scrum proposes a Sprint Planning which is a meeting divided in two parts: in the first part all stakeholders (customer, scrum master and developer team) selected from product backlog the items that should be worked in the Sprint. In the second part, the team discusses technical issues related to selected items. Crystal methods define a planning meeting to decide the next increment of the system. In FDD, development is feature-oriented which includes the creation of a high-level plan where features are organized according to their priority. Before each iteration, customer and team decide together which features should be developed in the next iteration. In DSDM the scope of an iteration is planned beforehand. Planning the cycles in ADS is part of the iterative process. ADS method also proposes *Joint Application Development* (JAD) sessions which are workshops where developers and customer representative discuss product features and decide the components that will be included in the next cycle. The practice *Iteration Plan* in OpenUp suggests planning an iteration in detail only when it is due to start. An iteration planning meeting should be hold by the whole project team who decide the iteration scope.
Industry Perspective:	All the participants considered the Iteration Planning relevant. One participant mentioned that in some Projects the Iteration Planning is divided in two parts (different from Scrum division mentioned above). In the first part (which the participant called Pre-Iteration Planning) a team member representative (usually the most experienced) and a business analyst discuss details related to items that are candidates to be developed in the next iteration. The goal of the first part is to anticipate potential technical problems (inconsistent business rules, incomplete or hard-to-implement requirements, business processes not mapped, etc). In case any problem is detected, the issue should be resolved before the Iteration Planning.

Evaluation Strategy:	We identify three alternatives to verify the presence of the Iteration Planning. One alternative is to check if there is any task in the iteration plan that represents the Iteration Planning meeting. Another alternative is to verify if there is any task in the iteration plan that produces an Iteration Plan artifact. A third alternative is check if all the tasks in the iteration plan are estimated before start the iteration.
Parameters:	No parameter was identified for the rule.

4.8 Rule 08 - Avoid Complex Tasks

The rule states that complex tasks should be avoided. During iteration planning, the team should try to break complex tasks in simpler tasks.

Agile Methods:	The motivation for *Avoid Complex Tasks* rule in Scrum is derived from a technique for project management called Burndown Chart. A Burndown Chart reflects the daily progress of the team and decreases according to the number of finished tasks. The chart is expected to decrease daily after Daily meeting. Otherwise a delay in working-in-progress is detected. The problem with complex tasks - those whose duration exceeds 8 hours - is that they may give a false indication that the work-in-progress is not evolving. On the other hand, simple tasks make sprint management easier and more reliable since it is expected that each developer to finish at least one task per day. In FDD features should be small enough to be implemented in a few hours or days.
Industry Perspective:	All participants indicated - with different levels - that the rule is relevant. No additional comment was given.
Evaluation Strategies:	An approach to evaluating *Avoiding Complex Tasks* is to verify whether the tasks estimations are above the allowable limit.
Parameters:	A *Maximum Estimation Allowed* parameter could be used to configure the rule.

4.9 Rule 09 - Avoid Unplanned Work

Agile teams usually work with tight deadlines. During the iteration planning they commit to delivery some parts of the software in a given time. To achieve the agreed commitment, teams have to work without interferences following the iteration plan. Since unplanned work represents tasks included in the iteration plan during iteration (after iteration planning), it should be avoided because it compromises the commitment with iteration goal.

Agile Methods:	*Avoid Unplanned Work* is mentioned by Scrum, DSDM and OpenUp. In Scrum, the Sprint Backlog is stable and unchangeable. DSDM method argues that in order to protect the agreed end date of an Iteration, low priority and unplanned tasks may be outside the scope of delivery. In OpenUp, once an iteration is under way, it should be allowed to proceed according to its plan, with as little external interruption as possible.
Industry Perspective:	One participant said that indicators of unplanned work are important in his organization. The indicators can be used for many purposes such as: (a) To evaluate iteration planning quality. Unplanned work could indicate that expected tasks were not included in the iteration plan. The team members may not know properly the process or they were careless in creating the plan. (b) To detect scope changing or iteration goal deviation.
Evaluation Strategies:	One alternative to identify unplanned work is checking whether tasks have been created during the iteration. Other alternative is using a specific field to categorize tasks as planned/unplanned.
Parameters:	A *Maximum Unplanned Work Tolerance* parameter could be used to define the threshold of unplanned work that is tolerated in an iteration.

4.10 Rule 10 - Avoid Dependence on Internal Specialists

A desired agile team is one in which all participants can work on any feature. The team should avoid that a member become the only specialist in a feature or technology in order to not become dependent on that member.

Agile Methods:	The rule was mentioned by two methods: XP and Scrum. The XP practice *Collective Ownership* states that anyone can change any part of the code at anytime. Scrum mentions the problem of dependency on internal experts and encourages the team to become homogeneous in their abilities. When a team identify that a member does not have skills required to a task, he/she became a natural candidate to perform the task. Scrum also mentions the problem of dependence on external specialists - which happens when the team depends on a specialist that is not part of the team. Although important, *Avoiding Dependence on External Specialists* was not included in the catalog since it was not considered a rule prone to formal validation.
Industry Perspective:	There was no unanimity between participants. While the majority considered the rule relevant - in different levels - a few of then said the rule is not relevant at all. No additional comment was given.
Evaluation Strategy:	One way to identify a possible threat to the rule is assessing the number of workers that have been working on the same feature. If just one worker is performing all tasks from a feature, it could be an indication that the project is becoming dependent on that developer.
Parameters:	A *Feature Complexity Boundary* parameter could be used as trigger to enable the verification of the rule. So, simple features could be developed by the same worker and complex features (those which exceed the configured parameter) should have more than one developer working on it.

5 Discussions

The research identified an initial set of 20 agile software development rules that was object of a survey that aims at characterizing the rules according to their relevance. It was observed that most of the rules were considered relevant according to the personal opinion of the survey participants. The exceptions were the rules *Avoid Shared Developer*, *Every Iteration should be the same length* and *Development Team should be small* Teams that were considered not relevant at all. Among the main arguments, participants said that these rules are very prescriptive or affect important organizational policies.

Regarding the relevance of rule's formal validation, it was also observed that those rules that depend on individual workers are more prone to be considered important to formal verification. For example, the rule *Every Iteration should have an Iteration Planning*, besides it has been mentioned by all agile methods analyzed, its formal verification was considered not relevant. Participants argued that *Iteration Planning* is so incorporated into their process and depends on almost the whole team that its formal verification is not necessary at all. However, the formal verification of the rule *Higher Priority Tasks should be performed first* - that was mentioned by only two methods - was considered relevant by all participants. They argued that the choice of which task should be selected from Iteration Backlog usually depends on worker's

individual decision and therefore a formal verification could be important to avoid non-compliances. Other rules that typically depend on individual workers include *Avoid Dependence on Internal Specialists* and *Avoid Unplanned Work*.

As the threats to the validity of this study we can mention:

- The identification of software development rules and the assessment of what rule is prone to formal validation were made initially by the authors. In order to mitigate this threat, the first results were submitted to a survey with specialists in software project management and quality assurance;
- The survey was conducted with a sampling that is not representative enough to allow us to affirm that the set of rules identified represent the most relevant rules;
- It was also observed in the survey that participants tend to consider rules where they are directly responsible for compliance less relevant to formal validation. For example, the rule *Development Teams should be small* was considered relevant, however participants considered the formal validation of this rule no relevant. One possible threat to validity of the survey is that people tend to underestimate the importance of formal validation for rules that they are responsible for.

6 Related Work

There are many studies that investigated agile methods practices and characteristics. Abrantes et al [28] conducted a study to identify the most commonly used agile practices. The authors identified a set of 12 practices: *test driven development*, *continuous integration*, *pair programming*, *planning game*, *onsite customer*, *collective code ownership*, *small releases*, *metaphor*, *refactoring*, *sustainable pace*, *simple design* and *coding standards*.

Miller [29] also proposed a set of characteristics to agile processes that includes: *modularity on development process*, *iterative with short cycles*, *time-bound with iteration cycles from one to six weeks parsimony in development processes removing unnecessary activities*, *adaptive with possible emergent new risks*, *incremental process approach that allows functioning application building in small steps*, *convergent (and incremental) approach* and *people-oriented*.

Although many others researches that aimed at mapping agile practices have been conducted [30] [31] [32] [33] [34] [35], for the best of our knowledge, there is no study that investigated agile methods from the perspective of this study: trying to identify a set of agile software development rules that are relevant and prone to formal validation.

7 Conclusions and Future Works

The research identified an initial set of 20 agile software development rules that was subject of a survey that aims at characterizing the rules according to their relevance. The 10 most relevant software development rules were presented in a structured catalog. For each rule identified, the catalog presents: title and description, which agile

method mentioned the rule, the industry perspective and strategies for formal verification and customization.

Although the software development rules have been extract from agile practices and characteristics, the catalog is not intended to describe practices and characteristics of agile methods. In fact, the catalog aims at presenting a set of software development rules that were considered relevant and whose formal validation could be important to software projects. Thus, the catalog can be used to someone who intend to implement a computational system that supports the assessment of software development rules for projects based on agile methods. Making software development rules explicit through a structured catalog is a prelude for a formal representation that could be ultimately used by an automatic and on-the-fly compliance checking solution.

It is important to note that many software development rules extracted from agile methods documentation were not included even in the initial version of the catalog. The discarded rules were considered not prone for formal verification typically because the information needed to checking the rule is not explicitly represented.

As future work, we can mention: (a) extending the catalog including software development rules present in traditional (or non-agile) processes; (b) identifying the most relevant rules through a more comprehensive survey with representative sampling.

References

[1] R. Vaculin, R. Hull, T. Heath, C. Cochran, A. Nigam, P. Sukaviriya, Declarative business artifact centric modeling of decision and knowledge intensive business processes, in: 2011 IEEE 15th International Enterprise Distributed Object Computing Conference, Institute of Electrical and Electronics Engineers (IEEE), 2011.

[2] C.D. Ciccio, A. Marrella, A. Russo, Knowledge-Intensive Processes: Characteristics Requirements and Analysis of Contemporary Approaches, Journal on Data Semantics. (2014).

[3] N. Chen, S.C.H. Hoi, X. Xiao, Software process evaluation: A machine learning approach, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011: pp. 333–342.

[4] X. He, J. Guo, Y. Wang, Y. Guo, An Automatic Compliance Checking Approach for Software Processes, in: 2009 16th Asia-Pacific Software Engineering Conference, 2009.

[5] A. Fuggetta, Software process, in: Proceedings of the Conference on The Future of Software Engineering - ICSE 00, ACM Press, 2000.

[6] P.H. Feiler, W.S. Humphrey, Software process development and enactment: concepts and definitions, in: [1993] Proceedings of the Second International Conference on the Software Process-Continuous Software Process Improvement, Institute of Electrical and Electronics Engineers (IEEE), n.d.

[7] J. Münch, O. Armbrust, M. Kowalczyk, M. Soto, Software Process Definition and Management, Springer Berlin Heidelberg, 2012.

[8] I. Sommerville, Software Engineering, 9th ed., Addison-Wesley, Harlow, England, 2010.

[9] R. Bendraou, M.-P. Gervais, A Framework for Classifying and Comparing Process Technology Domains, in: International Conference on Software Engineering Advances (ICSEA 2007), Institute of Electrical and Electronics Engineers (IEEE), 2007.

[10] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R.C. Martin, S. Mellor, K. Schwaber, J. Sutherland, D. Thomas, Manifesto for Agile Software Development, Manifesto for Agile Software Development. (2001). http://www.agilemanifesto.org/.
[11] A. Cockburn, Agile Software Development, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[12] W. Cunningham, Extreme Programming, (n.d.). http://wiki.c2.com/?ExtremeProgramming.
[13] S. Alliance, Learn About Scrum, (2016). https://www.scrumalliance.org/why-scrum.
[14] J.A. Highsmith III, Adaptive Software Development: A Collaborative Approach to Managing Complex Systems, Dorset House Publishing Co., Inc., New York, NY, USA, 2000.
[15] J. Stapleton, Dynamic Systems Development Method: The Method in Practice, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
[16] J.D. Luca, Feature Driven Development FDD, (n.d.). http://www.featuredrivendevelopment.com/.
[17] E. Foundation, OpenUp Methodology, (2012). http://epf.eclipse.org/wikis/openup/.
[18] P. Abrahamsson, O. Salo, J. Ronkainen, J. Warsta, Agile software development methods - Review and analysis, VTT PUBLICATIONS, 2002.
[19] K. Beck, Embracing change with extreme programming, Computer. 32 (1999) 70–77.
[20] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
[21] K. Schwaber, SCRUM Development Process, in: Business Object Design and Implementation, Springer London, 1997: pp. 117–134.
[22] K. Schwaber, M. Beedle, Agile Software Development with Scrum, 1st ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
[23] M. Berteig, Rules of Scrum, (2015). http://www.agileadvice.com/rules-of-scrum/.
[24] S.R. Palmer, M. Felsing, A Practical Guide to Feature-Driven Development, 1st ed., Pearson Education, 2001.
[25] A. Bhamani Kajornboon, Using interviews as research instruments, (2005).
[26] S. Oishi, How to Conduct In-Person Interviews for Surveys, SAGE Publications Inc., 2003.
[27] A.N. Ghazi, K. Petersen, sri sai vijay raj Reddy, H. Nekkanti, Survey Research in Software Engineering: Problems and Strategies, (2017).
[28] J.F. Abrantes, G.H. Travassos, Common Agile Practices in Software Processes, in: 2011 International Symposium on Empirical Software Engineering and Measurement, 2011.
[29] G.G. Miller, The Characteristics of Agile Software Processes, in: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39), IEEE Computer Society, Washington, DC, USA, 2001: pp. 385–.
[30] X. Yu, S. Petter, Understanding agile software development practices using shared mental models theory, Information and Software Technology. 56 (2014) 911–921.
[31] T. Dingsøyr, S. Nerur, V.G. Balijepally, N.B. Moe, A Decade of Agile Methodologies, J. Syst. Softw. 85 (2012) 1213–1221.
[32] S.C. Misra, V. Kumar, U. Kumar, Identifying Some Important Success Factors in Adopting Agile Software Development Practices, J. Syst. Softw. 82 (2009) 1869–1890.
[33] V. Subramaniam, A. Hunt, Practices of an Agile Developer: Working in the Real World, Pragmatic Bookshelf, Raleigh, NC, 2006.
[34] O. Ktata, G. Lévesque, Agile Development: Issues and Avenues Requiring a Substantial Enhancement of the Business Perspective in Large Projects, in: Proceedings of the 2nd

Canadian Conference on Computer Science and Software Engineering, ACM, New York, NY, USA, 2009: pp. 59–66.
[35] R.C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003