TWINS – This Workflow Is Not Scrum: Agile process adaptation for Open Source Software projects

Paul T. Robinson
Sony Interactive Entertainment LLC
Amherst MA, USA
paul.robinson@sony.com

Sarah Beecham
Lero – the Irish Software Research Centre
University of Limerick, Ireland
sarah.beecham@lero.ie

***Abstract*—It is becoming commonplace for companies to contribute to open source software (OSS) projects. At the same time, many software organizations are applying Scrum software development practices, for productivity and quality gains. Scrum calls for self-organizing teams, in which the development team has total control over its development process. However, OSS projects typically have their own processes and standards, which might not mesh well with a company's internal processes, such as Scrum. This paper presents an experience report from Sony Interactive Entertainment (SIE), where the "toolchain CPU compiler" team directly participates in the "LLVM" OSS project. The team ran into a number of difficulties when using Scrum to manage their development. In particular, the team often failed to complete Scrum sprints where tasks required interaction with the open source community. We look at how the team redefined task flows to alleviate these difficulties, and eventually evolved a highly modified process, dubbed TWINS (This Workflow Is Not Scrum). We assess the revised process, and compare it to other established agile methods, finding it bears a strong resemblance to Scrumban (the SIE team was not aware of Scrumban previously). The TWINS framework presented here may help other organizations who develop software in-house and engage in OSS projects, to gain the best of both worlds.**

Keywords—agile software development, Scrum, open source software, Kanban, Scrumban

I. INTRODUCTION

It is becoming commonplace for companies to engage in open source software (OSS) development, reinforcing the statement by the Eclipse Foundation's executive director that "every software company is an open-source company" [1]. At the same time, companies developing software are widely adopting agile frameworks, particularly Scrum [2, 3].

A comparison of agile and OSS development [4] shows that they share many underlying principles such as an emphasis on highly skilled individuals working in self-organizing development teams. Change is embraced, and supported by short feedback loops, frequent releases of working code, and close collaboration with customers. The proliferation of agile methods (Scrum, Kanban, Extreme Programming) tells us, however, that consistent underlying principles do not imply consistent practices. Companies participating in OSS projects find that they must adapt to, or at least coordinate with, "work practices that are appreciated by community members" [5] even if both conform to agile principles

This intersection of the agile and OSS trends—specifically, a company engaging with an OSS project while using agile processes internally—requires some adaptation on the agile side. Low levels of engagement can be handled simply [6], but adaptations for higher levels of engagement are not particularly simple. We describe how the toolchain CPU compiler [1] development team at Sony Interactive Entertainment (SIE) R&D[2] started using Scrum to work on the LLVM[3] project [7], a widely used OSS compiler. We cover the process difficulties encountered with respect to interacting with the LLVM project, some attempted revisions to manage the interactions, and finally the revised process developed to address these problems, dubbed "This Workflow Is Not Scrum" (TWINS). We assess the TWINS process and contrast it with a close, well-known relative, Scrumban [8].

The paper is organized as follows. In Section II we provide a background on agile methods, and how agile relates to open-source projects. Section III provides a context for this experience report, describing the open-source project, the SIE project, and the SIE team's process goals. Section IV describes the inherent conflict between Scrum and the LLVM project process. Section V details some modifications to Scrum to address this conflict, while Section VI describes the rather different TWINS (This Workflow Is Not Scrum) process that the SIE team finally adopted. Section VII assesses the success of TWINS, Section VIII compares TWINS to other agile workflows, and Section IX summarizes our paper.

II. AGILE METHODS AND RELATED LITERATURE

This section briefly covers the relevant agile principles and methodologies referred to throughout the paper, followed by a review of the literature on agile processes as used in conjunction with OSS projects.

A. Agile values and frameworks

Four values form the foundation for the agile manifesto: Individuals and interactions over processes and tools; working software over comprehensive documentation; customer collaboration over contract negotiation; responding to change over following a plan. These four values are applied in different ways in each of the many agile methodologies, but it is these guiding principles that binds them together. With organizations becoming more global, and also scaling through

[1] The GPU compiler is produced separately and is not based on LLVM.
[2] This is not the only compiler team within SIE that works with LLVM, and other teams use other processes.
[3] LLVM is not an acronym, it is simply the project's name.

978-1-7281-3393-5/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSSP.2019.00014

acquisitions or mergers [9], we have seen the emergence of many scaling agile frameworks [10, 11]. Since our report draws on Scrum, Kanban and Scrumban, we give a brief overview of these methods.

1) Scrum

Scrum emerged in the early 1990s as a means to manage work on complex products. Scrum is a framework onto which teams place their own processes and techniques [12]. The Scrum framework is made up of Scrum Teams that have well defined roles, and ceremonies, artifacts and rules. "Each component within the framework serves a specific purpose and is essential to Scrum's success and usage" [12]. Scrum is underpinned by a set of five values: commitment, courage, focus, openness and respect. The details behind the roles, events, and artifacts, that govern relationships and interactions between them are discussed in our results (Section VIII) that compares TWINS to the Scrum, Kanban and Scrumban frameworks.

2) Kanban

In his foreword to Kniberg and Skarin's book [13] (that compares Scrum with Kanban), David Anderson (an advocate, promoter and coach of the Kanban methodology), observes that many organizations were quick to adopt Kanban because they were struggling with Scrum. Kanban is based on the simple notion that there should be minimal Work In Progress where nothing new is started until there is capacity: i.e. a given task is completed or moved downstream.

3) Scrumban

As its name suggests, Scrumban captures practices from both Scrum and Kanban to create a framework for managing software engineering practices. Scrumban as a term first appeared in 2008, in the essays by Ladas collected in [14], who recommended Scrumban for teams already using Scrum who want a more flexible way of developing software, with the benefit of transparency into the process, speed for decision making, and fewer rules. The literature has very few recorded experiences of Scrumban in a purely industrial context. Examples tend to be either based on applications in the university coding factories and labs, e.g. [15, 16], or on how Scrumban is used as an alternative to Scrum to reduce the workload when integrated with a user centered design methodology [17]. However, some of these studies do help us to develop a better understanding of how and why teams transition from Scrum to Scrumban. For example the well-researched empirical work by Banijamali et al. [16] highlighted benefits and limitations, and a real implementation of Scrumban in an SME by Yilmaz and O'Connor [18] found that combining Scrumban with gamification enhanced individual and organizational productivity. As by Banijamali et al, Scrumban emphasizes work breakdown and the delegation of responsibilities: "Scrumban applies Scrum as a prescriptive method of team-work to complete the work, while it encourages process improvements through Kanban to allow projects to continuously optimize the processes and number of tasks" [16].

B. OSS and Agile

There is very little research outlining how in-house agile software development teams interact and collaborate with OSS projects. Diebold et al. [19] examine how companies adapt Scrum in a variety of ways and contexts, but none of these involved an OSS project. Dahlander and Magnusson [20] look at business-value interactions rather than development processes. Butler et al. [21] look at specific practices by companies participating in open source projects, primarily with respect to tools used, but do not consider their overall development processes. Paasivaara et al. [22, 23] examine scaling Scrum across geographies within a company, but not outside the company. Crowston et al. [24] summarize research on processes and practices within an open source project but not the contributing companies. Theunissen et al. [6] propose agile adaptations for low levels of OSS engagement but not for active contributors.

Several authors have researched how Scrum works in a variety of open source contexts. Some are from an educational standpoint, where Scrum is applied to manage student teams developing open source tools or collaborating on real-life OSS projects [25-27], while others simply use open source tools to support their agile development [28-32]. However, a handful of examples indicate that it is possible for industry to apply agile methods and develop software in conjunction with open source communities. An extreme example is reported by Gary et al. [33], who appear confident that agile methods can be adapted and used in a safety critical context, despite development being open source. These authors give examples of the lightweight agile approach used to develop a software application that is now used in hospital settings, noting that this is just a start and further work is needed. Nearly all experience reports on this subject note that agile methods, and Scrum in particular, need to be adapted to fit the OSS context. For example, Düring [34] found sprints particularly challenging explaining that, "techniques such as "sprinting" has been a core balancing act for the project since its inception. "Sprints" in the Python community differs from the Scrum version of sprints". The closest work to our own is that of Rahman et al, who have developed a model for Scrum for OSS, but do not appear to have validated the model in a real project [35].

III. Experience Report Context

This section provides background information on the LLVM open source project and its processes, as well as the SIE development team's project and its process goals.

A. OSS Project: LLVM

The LLVM Project [36] "is a collection of modular and reusable compiler and toolchain technologies." That is, the overall project consists of a number of sub-projects that, among them, provide: *llvm*, a library of optimizers and code generators for a variety of target architectures; *clang*, a C/C++ compiler front-end using the LLVM optimizer and code generators; *libc++*, a C++ standard library; *lld*, an object-file linker; *lldb*, a debugger; and other related tools and libraries. LLVM components are widely used commercially, by other OSS projects, and for academic compiler research.

LLVM began as a research project at University of Illinois at Urbana-Champaign [37] and has since received the 2010 SIGPLAN Programming Languages Software Award and the 2012 ACM Software System Award. The *llvm* component has

grown to 2.37 million lines of code (MLOC) while the *clang* frontend has 1.61 MLOC[4] with hundreds of contributors; data from 2015 shows these components received an average of 50 commits to the public repository per day [38].

The LLVM Foundation [39] was created in 2014 to provide fiscal and operational support to the community of LLVM developers. The Foundation organizes semi-annual conferences in California and Europe for the benefit of the community; it also sponsors other activities, as well as providing an independent legal entity to be the owner and operator of supporting infrastructure (servers, mailing list host, etc.). The Foundation explicitly does not control software development policies and procedures, apart from a Code of Conduct [40]; the developer community sets its own development processes.

The LLVM project creates a formal software release twice a year. The source code is captured on a separate development branch, and subjected to more testing than usual for the primary (trunk) branch. Release branches are generally active for a relatively short time, 4-6 weeks, while bugs found by testing are fixed; after that, a final version of the software is built and packaged for use on various platforms.

B. LLVM Development Process

The LLVM project has a defined development process [7] summarized here.

LLVM, like most OSS projects, is developed publicly. In particular this means the source code, mailing lists, code-review website, and bug tracking system are available for anyone to read. Registering for anything short of commit access is simple, and permits full participation in the project.

LLVM does not define the notion of a *core developer*. (In many OSS projects, a core developer is someone with *commit access* [41], a role which typically comes with various other rights and responsibilities. LLVM is very liberal with granting commit access, requiring only a short history of accepted patches, so this is not an important distinction in the LLVM community. However, LLVM does define the notion of a *code owner*, an individual who takes responsibility for the overall direction of a given area within the project and also for ensuring that patches within that area are reviewed. Code-owner status indicates commitment to long-term success of the project, and some degree of architectural authority over their particular area.

Code reviews take place in public. A proposed source patch is posted on the project's review website or emailed to one of the project mailing lists. The author may designate one or more *reviewers*, and optionally one or more *subscribers*; the appropriate project mailing list is always subscribed, so the entire community has easy access to all reviews. Designating a particular reviewer or subscriber does not require participation by those people, it is merely a suggestion. Anyone may comment on a patch, and any proposed patch must be explicitly approved before the author may commit it. Strictly speaking, anyone may approve any patch; however, community practice is that approval ought to come from someone with relevant experience. Larger or more complex changes ought to have direct approval from the appropriate code owner or another developer with long experience in the area, and generally design changes should be presented and discussed on the mailing list before posting a patch.

In practice, code reviews are not guaranteed to be timely, and in rare cases can take an extremely long time [42]. Reviewers are commonly from a different organization than the patch author, and may have very strong opinions about how a particular change ought to be implemented; so, posting a finished implementation for review is no guarantee that the patch will be approved in a given amount of time or in anything like its original form.

C. Company Project: SIE Toolchain CPU Compiler

The SIE toolchain CPU compiler development team delivers the compiler and some related tools for the Sony PlayStation®4 developer toolchain [43]; the toolchain in turn is one component of the Software Development Kit (SDK) that Sony licenses to game development studios. The team has grown over time to have developers and QA engineers at three sites across California and the UK; in addition there is a project manager (acting as Scrum Master when the team was using Scrum), a Product Owner, and several line managers. (The company requested that we not state exact team numbers.) The compiler is based on the Clang compiler from the LLVM project, with minor modifications (on the order of 10 KLOC).

Using the common metaphor, the LLVM project is *upstream* of the SIE project, and conversely the SIE project is *downstream* of the LLVM project; that is, SIE bases its compiler on source from the LLVM project, and incorporates a continued flow of updates from the LLVM project. The team also refers to its internal copy of the development sources as being the *local repository*, which may contain local changes that are not in the *upstream repository*. If the team decides that a local change should be committed to the upstream repository, they say the change is *going upstream*.

The toolchain team releases compiler updates to the SDK twice a year. The management team in charge of the toolchain determined that these releases should be based on the upstream LLVM semi-annual releases, because the additional testing and bug-fixing done on the upstream releases provides valuable stability. The timing of the upstream releases and SDK releases are not well aligned, which means the SIE team must carry a local release branch (created to match the upstream release branch) for some time before producing their own release. Because of the customizations added by the SIE team, they do not use the binary release packages created by the LLVM project, but build their own release packages.

D. SIE Process Goals

As product developers, the SIE team naturally wants to deliver a robust, high-performance compiler with an array of useful features to their licensees (the game studios); and, to do so efficiently (within budget) and on schedule.

[4] https://www.openhub.net/p?ref=homepage&query=llvm retrieved 30 September 2018

However, as the team is leveraging an OSS project, there are additional process considerations. For example, when providing a bug fix to an OSS project, a contributor may be "drawn into a long discussion and additional work because the proposed fix uncovers a larger problem" [21]; or if a fully developed feature is rejected by the project, "the implementor now has to maintain the code, rather than the project community" [21] if the company has customers depending on it. Also, the LLVM project is evolving quickly; merging (integrating) the continual upstream changes into the team's local repository is ongoing overhead, which can be quite expensive [41]. Automating the merge helps a great deal but developers must manually resolve conflicts between local changes and changes made upstream. The team realized that contributions made directly to the upstream project necessarily avoided these conflicts. That is, team members should become direct LLVM project contributors and act as members of the LLVM community.

Team members therefore participate actively in the LLVM project, using the upstream mailing lists, code-review website, and so on. The cost of community participation mentioned in [21] is ameliorated by the savings in having fewer merge conflicts. Thus, working "upstream first" supports the goal of working efficiently (i.e., minimizing cost).

IV. Scrum versus the OSS Process

A. Team Use of Scrum

The SIE team initially took strong guidance from the standard Scrum descriptions (e.g. [44]); however, they did vary certain practices right from the start, as seems quite typical [19]. There were two major differences.

First, as mentioned in the previous section on Context, the team was not co-located. Discussing how the team handled their "distributed" process is beyond the scope of this paper; suffice to say that the common Scrum-of-Scrums approach (one Scrum team per site with separate backlogs) was rejected as being too divisive. To a first approximation, the different sites considered themselves to be one virtual team with a unified process and backlog, similar to [45].

Second, some responsibilities normally taken on by the developers were instead delegated to a backlog-management committee comprising the Scrum Master, Product Owner, and a development manager at each site. The engineers generated task estimates, but all backlog grooming and sprint construction tasks were performed by this committee.

B. The Inherent Conflict

Scrum depends on having a self-organizing and cross-functional development team. The Scrum Guide [12] says:

> *Self-organizing teams choose how best to accomplish their work, rather than being directed by others outside the team. Cross-functional teams have all competencies needed to accomplish the work without depending on others not part of the team*.

However, the LLVM project's development process is not under the control of the SIE team. Part of the team's work is "directed by others outside the team" and in particular, code reviews depend on "others not part of the team." Therefore, there is an inherent conflict between how Scrum is intended to operate and the requirements of the OSS development process.

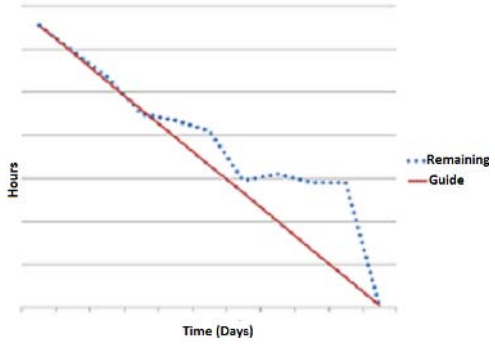

Fig 1: Burndown chart without upstream interaction

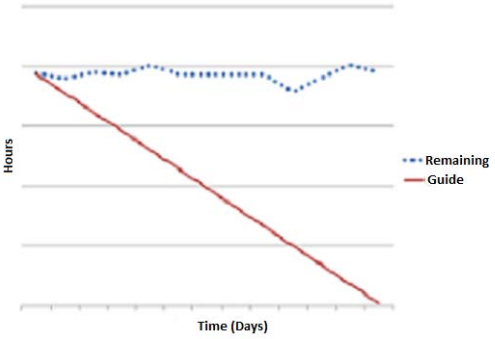

Fig. 2 Burndown chart with upstream interaction

This inherent conflict is confirmed by assessments of sprint success based on burndown charts. (At the company's request, specific numbers have been removed from these charts.) Early in the project there was less direct engagement with the LLVM project, with few tasks dependent on LLVM project code reviews, and sprints were generally successful (Fig. 1). Later, as more tasks required interaction with LLVM project members outside the company, sprints became much more problematic (Fig. 2) with remaining tasks never reducing or completing on time.

The problem, in a nutshell, is simply: How to make SIE's agile process able to cope with LLVM's open-source process?

V. Scrum Adaptation Attempts

Here we list some Scrum adaptations that the team tried, in order to alleviate the problems shown by the sprint failures without abandoning Scrum. Unfortunately the inherent conflict between Scrum and the LLVM process meant that all of these tactics failed to improve the situation.

1) Lengthen the sprint.

Revision: Change the length of the sprint from two weeks to four weeks.

Reasoning: Estimates for tasks *without* any upstream dependencies were fairly reliable; a scatter plot showing actual versus estimated effort (zero meaning a perfect estimate)

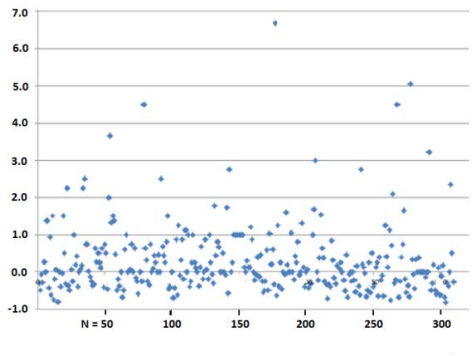

Fig. 3: Estimate accuracy: (actual - estimate) / estimate

showed that comparatively few tasks were severely underestimated (Fig. 3; N=309, mean=0.29, std.dev=0.93).

Result: Tasks *with* upstream dependencies still often failed to complete on time. The problem was not poor task estimates or over-long tasks, it was blockages due to the external dependency causing *elapsed* time (rather than effort) to run beyond the end of the sprint.

2) Commit locally, "send upstream" off-sprint

Revision: When a code change passed internal review, commit it locally and post it for upstream review. This counted as "done" for the task.

Reasoning: Because the review time was an external dependency and could not be reliably estimated, it could not be handled as part of the sprint. However, the code change had been committed locally and thus could be "done" as far as sprint completion was concerned.

Result: This tactic left the upstream review pending with little motivation on the team's part to follow it through. And, as the review process would typically cause anywhere from trivial to quite major changes in the implementation, the team then incurred an additional cost later on to resolve differences between the upstream implementation and the local version. This cost became a motivation to avoid sending anything upstream, which was directly contrary to the internal project goal to do as much as possible "upstream first."

3) Handle the review off-sprint

Revision: When a code change passed internal review, post it for upstream review but do not commit it locally. This counted as "done" for the task.

Reasoning: Because the review time was an external dependency and could not be reliably estimated, it could not be handled as part of the sprint. Not committing the change locally induced the developer to follow through with the upstream review.

Result: This tactic led to an unpredictable reduction in the team's capacity as reviews occurred. The developer would have to spend time responding to the review, but time spent on this task could not be charged to any on-sprint task. The unpredictable delay meant the feature would not necessarily be "done" in terms of delivering sprint functionality; even worse, no task was identified as required for completing the feature, making tracking feature completion more difficult.

4) Handle upstream review as its own task

Revision: When a code change passed internal review, post it for upstream review but do not commit it locally. This counted as "done" for the original task, and started a new "upstream review" task.

Reasoning: Despite the upstream review being an external dependency, managing it as a task would provide better tracking.

Result: Given that the upstream review process had not changed, dividing the Scrum tasks this way really did not help; it just changed which task was incomplete. Like the previous tactic, the feature was not "done" for purposes of delivering sprint functionality. As there was a hard dependency between completing development and starting the upstream review, Scrum principles required that the review task had to be in a subsequent sprint. In violation of Scrum principles, only the author of the change could take on the review task; Scrum wants anyone to be able to take any task.

5) Summary

Burndown data confirmed that sprints were not working as intended, and a handful of small tweaks to Scrum were clearly insufficient to address the problem. The inherent conflict between Scrum and open source had led to sprint after sprint of incomplete work. While this consequence was immediately plain to all the developers, it had a secondary consequence visible mainly to the backlog-management committee: New tasks continued to accumulate at the same time that old tasks failed to be completed (as evidenced by Fig. 2), making the full backlog unmanageably large.

Based on these failings, management determined that Scrum was not the most appropriate process for the project. Scrum was a means to an end (i.e., delivering the product on time, within budget, and with appropriate quality), not an end in itself. So, they made more radical changes to come up with a new and more appropriate development process.

VI. TWINS

This section describes the TWINS (This Workflow Is Not Scrum) process, describing its agility, roles, backlog and task management, and communication and introspection practices.

TWINS was designed to address the "inherent conflict" between Scrum and the open-source process as described in Section IV, without clinging to Scrum as in Section V. The name reflects that it uses a different set of practices from Scrum, although it adheres to agile principles (http://agilemanifesto.org/) and even retains a few roles and practices used by Scrum, as detailed later in this section.

Because the open source process is effectively non-negotiable and a hard dependency for doing work upstream, the company side of the interaction had to change to accommodate it. The primary outcome of this is that in TWINS, there is no fundamental notion of a sprint, but rather a more Kanban-like style of selecting and tracking tasks.

As a software development process, TWINS primarily considers how to identify the most important remaining work, and how to keep track of the work currently in progress. With the full backlog growing to unmanageable size, a single prioritized list was no longer feasible. TWINS organizes tasks into multiple backlogs, prioritized independently. The backlogs continue to be groomed by a backlog committee.

TWINS retains the view of developers as a single cross-site virtual team. All work is shared across the whole team. Communication remains regular, frequent, and easy. The following subsections describe the process in more detail.

A. Agility

While the fundamental agile principles as commonly expressed (http://agilemanifesto.org/) are not universally admired [46] many of them directly address process topics and so we examine these to evaluate the claim that TWINS is agile, even though it Is Not Scrum.

TWINS welcomes change by doing continual prioritization of the task backlog, similar to Scrum. TWINS targets delivery of working software with a continuous-delivery approach; completion of every task should in principle result in deliverable software, where Scrum periodically captures a set of tasks into an iteration that then represents a deliverable product. TWINS specifies a periodic retrospective ceremony modeled on Scrum's, although it is time-based rather than iteration-based.

B. TWINS Roles

The fundamental role in TWINS is the developer. The developers for the project form a single virtual team, even if they are under different managers or not co-located. For example, the SIE toolchain CPU compiler team has developers and QA engineers across three sites, split between California and the UK, and each site has a development manager.

TWINS has a Product Owner, like the Scrum role, who has final authority over product direction and feature content. The Product Owner and the development managers form the backlog management committee (retained from the team's Scrum modifications), who manage the product backlogs.

TWINS does not have the equivalent of a Scrum Master, since the backlog management committee generally takes on these responsibilities. Optionally there can be a Project Manager, who generally runs the meetings and is particularly responsible for the Retrospective.

TWINS does not specify how testing and quality assurance (QA) are managed. Scrum specifies that the development team is cross-functional and handles its own testing needs, whatever they are. The LLVM process similarly specifies that each functional change must come with a corresponding regression test, although these are typically narrowly focused and form only one part of an overall QA effort. The SIE team's experience is that superb developers are not necessarily superb testers; a separate team of QA professionals is highly valuable and provides an important extra level of verification. This experience is not universal [45].

C. Multiple Backlogs

Scrum presumes a single prioritized Product Backlog containing "everything that is known to be needed in the product" [12]. However, there are serious practical problems in attempting to manage a task list with hundreds of pending items. In the SIE team's case, there was also the consideration that some tasks were considered necessary for the upcoming release, and thus had an externally imposed deadline, but were not inherently higher priority than other, longer-term tasks planned for later releases. The relative timing of the upstream LLVM releases (determined by the LLVM project) and the corresponding SIE toolchain releases (determined by the SDK management) are not particularly aligned and certainly are not under the control of the compiler team. Therefore, both the main development branch and a release branch are nearly always active simultaneously.

TWINS addresses this by splitting the full backlog in two ways. First, there is a separate backlog for each planned upcoming release, plus one for a general "future" release. In SIE's case, this generally means one release backlog and the general backlog. Second, there is an unprioritized "other" collection of tasks, none of which are deemed important enough to be on either an upcoming release backlog or the general backlog. These tasks generally should not be discarded, as they typically represent very low severity bugs, or nice-to-have features, or the like.

D. Managing Pending Tasks

Tasks can appear for a variety of reasons, such as: a proposal for a new feature, from a customer or team member; a bug report from a customer; a test failure after integrating the next round of changes from upstream; or, some other source. Wherever a new task comes from, the backlog committee needs to determine how to handle it. (Generally some individual member of the committee handles it initially, subject to review by the committee.)

The first assessment is whether it belongs on the general backlog, a release backlog, or in the "other" pile; in some cases, a feature request might be outright rejected. If the task goes to the "other" pile, no more needs doing; otherwise, the task needs to be sized, and prioritized against the other tasks already on the chosen backlog.[5] Sizing, or estimating, may be done in a variety of ways; the SIE team used Planning Poker for a time, and currently uses "T-shirt sizes" (small, medium, large, extra-large) as a relatively coarse assessment. The former of course allows more precise-looking statistical analysis, while the latter is more qualitative. The general backlog, and any impending release backlogs, are prioritized independently.

At least weekly, the backlog committee reviews the release and general backlogs. This activity is primarily to verify that the backlogs are sorted appropriately, but also helps them keep an eye on the state of the release backlogs and whether progress on those tasks is on track to meet the release deadlines.

[5] In rare cases, of course, a critical bug from a customer will require an immediate "hot fix" outside the normal process.

Periodically, the backlog committee reviews the "other" pile of tasks, to see if any are worthy of moving to the general backlog, or conversely should be eliminated (perhaps a task has become obsolete, or has been implemented upstream by someone else). This review may be done when the general backlog is looking anemic, or on some schedule (perhaps quarterly) to see whether opinions about the worthiness of any "other" tasks has changed. This periodic review ensures that the "other" pile does not turn into a black hole where a task disappears, never to be seen again.

E. Working On Tasks

Developers needing a task to work on may choose a task from the general backlog or any release backlog. If a release deadline is approaching, the corresponding release backlog should take precedence. If a release backlog appears to be in danger of not having all tasks completed on time, managers may encourage developers to choose those tasks; in extreme cases, the backlog committee could temporarily close the general backlog.

While choosing the highest priority work is encouraged, a developer's particular skills or preferences might lead them to choose a task lower down on the list. Managers need to ensure that the higher priority tasks are in fact selected at some point, perhaps calling for a volunteer or using other encouragement.

A task moves through a sequence of states, modelled on Kanban [47]. Teams decide what set of states is most appropriate for their environment. In the case of the SIE team, patches intended for upstream might go through a sequence such as the following.

- Available: on a backlog and not assigned to anyone.
- Assigned: a particular developer is responsible for the task (although might not have started work).
- In Progress: the developer is working on the task.
- Local Review: when appropriate, a patch might go through a local pre-review before being posted for review upstream.
- Upstream Review: posted upstream for review.
- Fixed Upstream: accepted and committed upstream, but not yet integrated into the local repository.
- Done: the patch has been integrated into the local repository.

TWINS can use (but does not require) Kanban's work-in-progress (WIP) limit. At the individual level, TWINS assumes developers generally will not pull tasks unless they have capacity available; if someone has too many tasks, an Assigned or In Progress task may be "thrown back in the pool" of Available tasks. On a more global level, Kanban assumes that having many tasks in a given state indicates a blockage that the team can swarm on and resolve. TWINS is designed to work with external dependencies, and it is possible for a task to be "stuck" in a state for quite some time with essentially nothing that the team can do about. The most obvious case is the Upstream Review state; in one extreme case, a large change was ultimately broken into smaller parts and gradually accepted over the course of a full year. Another example is the Fixed Upstream state; ideally the integration of changes from upstream into the local repository is largely automated [38], but can be blocked because of unrelated merge conflicts or test failures that take time to resolve. This means that a WIP limit might occur that is outside the team's direct ability to resolve; by constraining those situations to particular task states, the team can still detect other WIP limit violations and address them as usual.

Small teams with small backlogs can manage their current tasks using the common method of "sticky" notes on a whiteboard or wall, demarcated into "swim lanes" indicating task status. As teams and backlogs grow, online tools such as Jira Agile[6] become appropriate. The key point is easy visibility into the status of active and pending tasks, not the technology used to display that status. However, each backlog should have its own display, as each backlog describes an independent set of tasks.

F. Delivering Working Software

Agile principles ask for working software to be delivered (or deliverable) frequently, both as a measure of team progress and to enable feedback from the customer(s). TWINS adopts the LLVM development process principle that every change needs to be buildable and testable; i.e., every change produces a working and at least potentially deliverable version of the software. The SIE team applies this principle to its automated merge processes as well, ensuring that changes from upstream will build and pass some basic tests before merging them into the local repository.

G. Communication and Introspection

TWINS adopts two useful practices for communication and introspection from Scrum, although practices that are keyed off of the sprint in Scrum are instead time-based.

The development team holds a daily Check-in meeting, comparable to the Daily Scrum meeting. Depending on time-zone and other considerations, there might be multiple Check-in meetings across different sites, or multiple sites might have a "virtual" meeting by teleconference or other medium. Ideally no more than 15 minutes, each developer gives a brief summary of their activity over the past day and projecting their activity for the next day. Blocking issues might be raised, but most discussion should be deferred until after everyone has done their Check-in; post-meeting discussion does not count toward the 15-minute limit. Terse "minutes" of the Check-in are logged on a web page available to everyone, to facilitate cross-site communication.

The entire team participates in a periodic Team-Wide Meeting, generally held monthly. At a minimum this meeting should itemize recently completed work, discuss plans for upcoming releases, and include a Retrospective (akin to Scrum's sprint retrospective). This meeting is also an opportunity for team members to give short presentations about new feature work or other topics relevant to the team. To accommodate the SIE team's geographical distribution, they hold this meeting as a videoconference call.

[6] https://www.atlassian.com/agile

As there are no sprints or other true iterations in TWINS, there is no corollary to the Sprint Planning meeting. Prioritization is done regularly (at least weekly) by the backlog committee, as described earlier.

Team communications are not limited to the Check-in and Team-Wide Meeting. The online task management system, online code reviews, email, and Internet Relay Chat (IRC) such as Slack are commonplace and at the discretion of the team. The team's communication practices are examined in more detail in [48].

VII. Effectiveness of TWINS

A. Methods

As TWINS does not have sprints we cannot measure its success with sprint artifacts. However, we can look at measures of product quality and team satisfaction with the process before and after the SIE team's transition from Scrum to TWINS

To measure product quality, we started with the number of bugs reported by licensees (customers) per semi-annual SDK release; at the company's request to obscure raw data, we scaled this data by the number of downloads of each SDK to generate a representative per-capita metric. Duplicate reports from multiple licensees are counted as separate bugs.

For team satisfaction, we looked at comments logged during the retrospective meetings. Retrospectives record five classes of comments from the team members: things that went well, things that did not go so well, suggestions, puzzles, and "learnings." The first two best represent the sentiment (or satisfaction) of the team, so we examined the number of each class of comment, per retrospective meeting, comparing the numbers before and after the switch from Scrum to TWINS.

B. Product Quality

Comparing releases using Scrum and releases using TWINS (Fig. 4), we see that the long-term quality trend is very similar (m= -0.22 for both; Scrum R^2=0.09, TWINS R^2=0.16).

While the data is limited, we conclude that the process transition had no significant effect (positive or negative) on the quality of the delivered product.

C. Team Satisfaction

Positive comments (Fig. 5) and negative comments (Fig. 6) both show a clear decline over time while using Scrum. This could indicate a sort of "comment fatigue" with overall responses dropping, but the near-identical slopes of the trend lines (-0.17 for positive comments, -0.20 for negative) suggest that overall sentiment did not particularly change during this period. By contrast, positive comments had a small but fairly clear increase when using TWINS, while negative comments had a slight decline. This divergence suggests improved sentiment during the TWINS period.

We conclude that team satisfaction overall improved after the switch to TWINS.

VIII. TWINS Compared to Other Agile Frameworks

The ceremonies in TWINS are closed and entirely in-house, since the SIE team follow a separate process from the

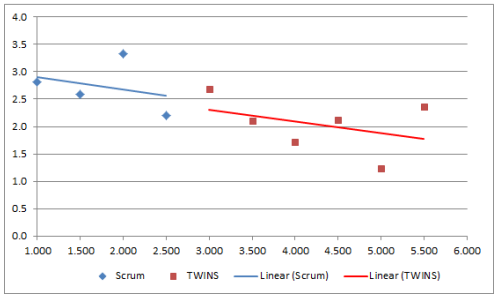

Fig 4. Bugs per 1000 downloads over time (SDK version)

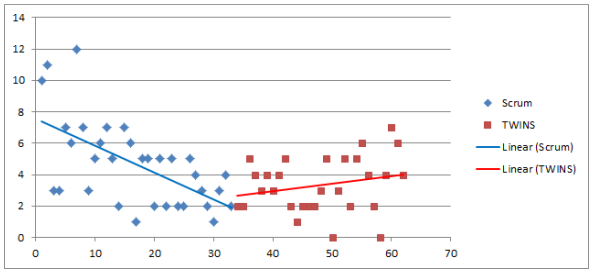

Fig 5. Positive comments over time (retrospective #)

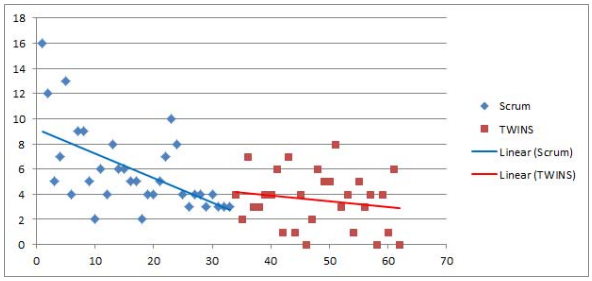

Fig 6. Negative comments over time (retrospective #)

OSS project. The only artefact that is shared is the upstream code. In common with Scrumban, TWINS has four ceremonies: the daily stand-up (attended by all developers), the monthly retrospective (attended by the entire team), the weekly review of release and general backlogs (attended by the backlog management committee), and the quarterly review of the "other task" pile (done by the Product Owner and possibly members of the backlog management committee). TWINS artefacts include the task status board (which developers primarily use just to select a new task) and the records of the stand-up and retrospective meetings.

Banijamali et al [16], compared the three frameworks (Kanban, Scrum and Scrumban), noting their similarities and differences. In Table 1, we reproduce and adapt this summary table, and add a "TWINS" column, highlighting the differences from Scrumban.

TWINS, like Scrumban, is a melding of Scrum and Kanban, differing primarily in adopting continuous delivery rather than iteration-based delivery, and making explicit work-in-progress (WIP) limits optional. Continuous delivery allows smoother integration of changes from the upstream OSS

TABLE I. SIMILARITIES AND DIFFERENCES AMONG KANBAN, SCRUM, SCRUMBAN, AND TWINS (ADAPTED FROM [16])

Kanban	Scrum	Scrumban	TWINS
No predefined roles for members	Predefined roles of Scrum Master, Product Owner and team members	Predefined roles of Scrum may vary within project time	Predefined roles of Product Owner and team members
Continuous delivery	Time-boxed sprints	Task board-based iterations	***Continuous delivery***
WIP limits amount of work	Sprint limits amount of work	WIP limits amount of work	***No explicit limits to amount of work***
Changes can be made at any time	No changes allowed mid-sprint	Changes allowed mid-sprint	Changes can be made at any time
Earlier planning and documentation necessary	Planning done after each sprint	Planning on demand, also within sprints	Planning on demand (no sprints)
Kanban board is persistent	Scrum board is reset after each sprint	Scrumban board is persistent	TWINS board is persistent
Size of task is not limited	Size of task limited to a sprint	Size of task is not limited	Size of task is not limited
Pull-based work management	Sprint backlog-based work management	Pull-based work management	Pull-based work management

project, and does not impose an artificial iteration deadline on tasks that may incur a long upstream review cycle. While WIP limits can be useful, with an OSS project the SIE team found that most WIP-limit issues come from external dependencies, and cannot be resolved by swarming or other local tactics; therefore the SIE team does not use them.

The SIE team was not aware of Scrumban at the time they developed TWINS, yet developed a strikingly similar workflow.

IX. SUMMARY AND FUTURE WORK

To date guidance on how organisations developing in-house software can successfully participate actively in open source projects has been largely theoretical. In this paper we highlighted how an agile team adapted their processes to ensure their development cadence meshed with the more fluid rhythm of the external open source project.

We presented an experience report showing how a team of developers at Sony Interactive Entertainment encountered an inherent conflict between their Scrum process, which assumes the local team has total control, and the open-source environment of the LLVM project, which requires ceding control of some activities to the open-source community. Following the Scrum paradigm of introspection and refinement, the team attempted various adaptations, none of which solved the inherent conflict. Ultimately the team developed and adopted a new process, calling it TWINS (This Workflow Is Not Scrum), which uses a continuous-flow model to better accommodate the interactions with open-source.

We performed a brief analysis of TWINS, showing that it has had no apparent effect (positive or negative) on product quality, while the developers seem happier with it than Scrum. A comparison with other well-known agile processes shows that TWINS is very similar to Scrumban, differing primarily in using continuous delivery rather than time-boxed iterations, and not applying explicit work-in-progress limits. This similarity arose even though the SIE team was not aware of Scrumban previously. The similarity lends confidence that the TWINS process would be readily applicable to other companies who want to use agile methods while actively contributing to an open-source project.

This independent development of a Scrumban-like process suggests that additional investigation of agile/open-source interactions would be fruitful. Are there other agile processes or adaptations that provide better OSS project interaction? In particular it would interesting to see whether "straight" Scrumban (likely using the "continuous flow" originally proposed by Ladas in [14], rather than time-boxed iterations) would be equally effective.

There is one unexamined assumption on the part of the SIE team, leading to this question: Is the "upstream first" goal really the most cost-effective tactic for handling integration of upstream and local changes? The SIE team accepted that the "upstream first" approach to engaging an OSS project would overall be more efficient, based on some early poor experiences with other tactics [38]. However, this was not evaluated with a proper cost/benefit analysis. Intuitively the most efficient tactic would differ based on a number of factors; research in this area should greatly benefit other companies attempting to make similar decisions.

ACKNOWLEDGMENT

P.T.R. thanks Casper Lassenius for encouragement to write this paper, and Joshua Magee for providing some of the data analysis. The name TWINS was proposed by Jennette Hardy. PlayStation is a registered trademark of Sony Interactive Entertainment Inc. "PS4", "PS3", and "PS Vita" are trademarks of the same company. This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094.

REFERENCES

[1] Mike Milinkovich, *Open Collaboration: The Eclipse Way. Online (2017).* https://www.slideshare.net/mmilinkov/icse-2017-keynote-open-collaboration-at-eclipse *Last visited: 2018-08-22.* 2017.

[2] VersionOne 2018, *12th annual state of agile report. WWW page, accessed 2018-08-01. URL* https://explore.versionone.com/state-of-agile/. versionone-12th-annual-state-of-agile-report, 2018.

[3] Leo R. Vijayasarathy and Charles W. Butler, *Choice of software development methodologies: Do organizational, project, and team characteristics matter?* . IEEE Software 33, 5 (Sept 2016), pp.86-94., 2016. **33**(5): p. 86-94.

[4] Stefan Koch. *Agile principles and open source software development: A theoretical and empirical discussion* in *International Conference on Extreme Programming and Agile Processes in Software Engineering*. 2004. Springer.

[5] Jonas Gamalielsson and Björn Lundell, *Sustainability of open source software communities beyond a fork: How and why has the LibreOffice project evolved?* . Journal of Systems and Software, 2014. **89**: p. 128-145.

[6] Morkel Theunissen, Derrick Kourie, and Andrew Boake, *Corporate-, Agile-and Open Source Software Development: A Witch's Brew or An*

Elixir of Life?, in *Balancing Agility and Formalism in Software Engineering*. 2008, Springer. p. 84-95.
[7] LLVM Project, *LLVM Developer Policy. Online (2018). http://llvm.org/docs/DeveloperPolicy.html Last visited: 2018-08-22.* 2018.
[8] Ajay Reddy, *The Scrumban [R]Evolution: Getting the Most Out of Agile, Scrum, and Lean Kanban. Agile Software Development Series*. 2016: Addison-Wesley.
[9] Mohammad Abdur Razzak, John Noll, Ita Richardson, Clodagh Nic Canna, and Sarah Beecham. *Transition from Plan Driven to SAFe®: Periodic Team Self-Assessment* in *International Conference on Product-Focused Software Process Improvement*. 2017. Springer, Cham.
[10] C. Ebert and M. Paasivaara, *Scaling Agile.* IEEE Software, 2017. **34**(6): p. 98-103.
[11] Mohammad Abdur Razzak, Ita Richardson, John Noll, Clodagh Nic Canna, and Sarah Beecham. *Scaling agile across the global organization: an early stage industrial SAFe self-assessment* in *Proceedings of the 13th Conference on Global Software Engineering*. 2018. ACM.
[12] Jeff Sutherland and Ken Schwaber, *The Scrum Guide: The Definitive Guide to Scrum. November 2017. https://www.scrum.org /resources/scrum-guide Last visited: 2018-08-22.* 2017.
[13] H. Kniberg and M. Skarin, *Kanban and Scrum-making the most of both*. 2010: Lulu. com.
[14] Corey Ladas, *Essays on Kanban Systems for Lean Software Development*. Lean Series. 2009, Seattle WA: Modus Cooperandi Press.
[15] C. Plengvittaya and D. Sanpote. *Scrumban for teaching at undergraduate program: A case study from software engineering students, University of Phayao, Thailand* in *3rd International Conference on Digital Arts, Media and Technology, ICDAMT 2018*. 2018.
[16] A. Banijamali, R. Dawadi, M. O. Ahmad, J. Similä, M. Oivo, and K. Liukkunen, *Empirical investigation of scrumban in global software development*, in *Communications in Computer and Information Science*. 2017. p. 229-248.
[17] D. I. Sensuse, D. Satria, A. A. Pratama, I. A. Wulandari, M. Mishbah, and H. Noprisson. *Integrating UCD into Scrumban for better and faster usability design* in *2017 International Conference on Information Technology Systems and Innovation, ICITSI 2017 - Proceedings*. 2018.
[18] M. Yilmaz and R. V. O'Connor, *A scrumban integrated gamification approach to guide software process improvement: A Turkish case study.* Tehnicki Vjesnik, 2016. **23**(1): p. 237-245.
[19] Philipp Diebold, Jan-Peter Ostberg, Stefan Wagner, and Ulrich Zendler. *What do practitioners vary in using scrum?* in *International Conference on Agile Software Development*. 2015. Springer.
[20] Linus Dahlander and Mats Magnusson, *How do firms make use of open source communities?* Long Range Planning, 2008. **41**(6): p. 629-649.
[21] Simon Butler, et al. *An Investigation of Work Practices Used by Companies Making Contributions to Established OSS Projects* in *In 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2018)*. 2018.
[22] M. Paasivaara, C. Lassenius, and V. T. Heikkilä. *Inter-team coordination in large-scale globally distributed scrum: Do scrum-of-scrums really work?* in *International Symposium on Empirical Software Engineering and Measurement*. 2012.
[23] M. Paasivaara. *Adopting SAFe to Scale Agile in a Globally Distributed Organization* in *2017 IEEE 12th International Conference on Global Software Engineering (ICGSE)*. 2017.
[24] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins, *Free/Libre open source software development: What we know and what we do not know.* Computing Surveys, 2012. **44**(2): p. article 7.
[25] L. Lavazza, S. Morasca, D. Taibi, and D. Tosi, *Applying SCRUM in an OSS development process: An empirical evaluation*, in *Lecture Notes in Business Information Processing*. 2010. p. 147-159.
[26] D. Damian, C. Lassenius, M. Paasivaara, A. Borici, and A. Schröter. *Teaching a globally distributed project course using Scrum practices* in *2012 2nd International Workshop on Collaborative Teaching of Globally Distributed Software Development, CTGDSD 2012 - Proceedings*. 2012.
[27] M. E. F. Parker and Y. F. del Monte, *The Agile management of development projects of software combining scrum, kanban and expert consultation*, in *IFIP Advances in Information and Communication Technology*. 2014. p. 176-180.
[28] T. Popović, *Getting ISO 9001 certified for software development using scrum and open source tools: A case study.* Tehnicki Vjesnik, 2015. **22**(6): p. 1633-1640.
[29] J. A. Knapp, A. Gearhart, L. S. Kellerman, and L. Klimczyk, *The Pennsylvania Newspaper Archive: Harnessing an open-source platform to host digitized collections online.* IFLA Journal, 2018. **44**(2): p. 143-153.
[30] M. Eckhart and J. Feiner, *How scrum tools may change your agile software development approach*, in *Lecture Notes in Business Information Processing*. 2016. p. 17-36.
[31] E. F. Collins and V. F. De Lucena Jr. *Software test automation practices in agile development environment: An industry experience report* in *2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings*. 2012.
[32] F. R. Cotugno and A. Messina, *Adapting SCRUM to the italian army: Methods and (open) tools*, in *IFIP Advances in Information and Communication Technology*. 2014. p. 61-69.
[33] K. Gary, et al., *Agile methods for open source safety-critical software.* Software - Practice and Experience, 2011. **41**(9): p. 945-962.
[34] B. Düring, *Sprint driven development: Agile methodologies in a distributed open source project (PyPy)*, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2006. p. 191-195.
[35] S. S. M. M. Rahman, S. A. Mollah, S. Anirban, M. H. Rahman, M. Rahman, M. M. Hassan, and M. H. Sharif, *OSCRUM: A modified scrum for open source software development.* International Journal of Simulation: Systems, Science and Technology, 2018. **19**(3): p. 20.1-20.7.
[36] LLVM Project, *The LLVM Compiler Infrastructure. Online (2018). http://llvm.org Last visited: 2018-08-23.* 2018.
[37] Chris Lattner and Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation* in *Proc of the 2004 International Symposium on Code Generation and Optimization (CGO '04)*. 2004. Palo Alto, California, Mar.2004.
[38] Paul Robinson and Michael Edwards, *Living downstream without drowning. Online. http://llvm.org/devmtg/2015-10/#tutorial4 Last visited: 2018-08-24.* 2015.
[39] LLVM Foundation, *The LLVM Foundation. Online; http://llvm.org/foundation Last visited: 2018-08-24.* 2018.
[40] LLVM Foundation, *LLVM Community Code of Conduct. Online: http://llvm.org/docs/CodeOfConduct.html Last visited: 2018-08-24.* 2018.
[41] Andrea Bonaccorsi and Cristina Rossi, *Comparing motivations of individual programmers and firms to take part in the open source movement: From community to business.* Knowledge, Technology & Policy, 2006. **18**(4): p. 40-64.
[42] Kristof Beyls *Can reviews become less of a bottleneck?* Lightning Talk – LLVM 2018 2018.
[43] Paul T. Robinson, *Developer Toolchain for the PlayStation®4. Online. http://llvm.org/devmtg/2013-11/#talk8 Last visited: 2018-08-24.* 2013.
[44] Ken Schwaber and Mike Beedle, *Agile software development with Scrum*. Vol. 1. 2002: Prentice Hall Upper Saddle River.
[45] Pernille Lous, Paolo Tell, Christian Bo Michelsen, Yvonne Dittrich, Marco Kuhrmann, and Allan Ebdrup. *Virtual by Design: How a Work Environment can Support Agile Distributed Software Development.* in *IEEE 13th International Confercent on Global Software Engineering (ICGSE)*. 2018. Gothenburg, Sweden.
[46] Bertrand Meyer, *Making Sense of Agile Methods.* IEEE Software, 2018. **35**(2): p. 91-94.
[47] David J Anderson, *Kanban: successful evolutionary change for your technology business*. 2010: Blue Hole Press.
[48] Paul Robinson. *Communication Network in an Agile Distributed Software Development Team* in *ICSSP 2019 International Conference on Software and Systems Process co-located with International Conference of Software Engineering*. 2019. Montreal, Canada: ACM.