Management of Interdependencies in Collaborative Software Development

Cleidson R. B. de Souza[1,2] David Redmiles[1] Gloria Mark[1] John Penix[3] Maarten Sierhuis[4]

[1]*School of Information and Computer Science, University of California, Irvine*
Irvine, CA, USA

[2]*Departmento de Informática, Universidade Federal do Pará*
Belém, PA, Brasil

[3]*Computacional Sciences Division, NASA/Ames Research Center*
Moffett Field, CA, USA

[4]*Research Institute for Advanced Computer Science, NASA/ Ames Research Center*
Moffett Field, CA, USA

Abstract

In this paper we report results of an informal field study of a software development team conducted during an eight week internship at the NASA/Ames Research Center. The team develops a suite of tools called MVP, and is composed of 31 co-located software engineers, who design, test, document, and maintain the different MVP tools. We describe the formal and informal approaches used by this group to manage the interdependencies that occur during the software development process. Formal approaches are legitimated by the organization, whereas informal approaches emerge due to the needs of the developers. We also describe how the software development tools used by this team support these approaches and explore where explicit support is needed. Finally, based on our findings, we discuss implications for software engineering research.

1. Introduction

Software development is typically a collaborative activity in which experts from different domains work together to produce a software artifact. Indeed, formal and informal communication account for more than half of developers' time [21], and cooperative activities account for about 70% of this time [30]. Therefore, breakdowns in communication and coordination efforts constitute one major problem in software development [3].

One of the reasons that cooperative software development is difficult is the large number of interdependencies that occur. These include interdependencies among activities in the software development process, among different software artifacts, and finally, in different parts of the same artifact. One example involves the design document and the requirements specification—if the specification changes, the design normally needs to be changed as well. Another example involves dependencies among parts of the same artifact, such as program dependencies—syntactic relationships between the statements of a program that represent aspects of the program's control flow and data flow [22].

Software engineering has already identified the need to manage these interdependencies and has been developing formal approaches to deal with them. For example, software development processes describe, among other things, when each artifact should be created during the software development effort. Such processes would prescribe that the requirements specification to be created before the design document to minimize problems due to the dependency between these documents. Design techniques have also been developed. Examples of such techniques include information hiding [19], which tries to minimize dependencies in the implementation by using the concept of coupling, and design patterns [7], which give dynamic (runtime) program dependencies explicit representation as static program structures, making them easier to manage. In addition to formal approaches, software engineering tools have been built to support the management of interdependencies. An example is configuration management systems that deal with dependencies in the source code.

Informal approaches are also used to manage the interdependencies. These practices exist because no matter how formal and well-defined a process may seem, it will always be incomplete, and also because formal approaches have practical limitations [8]. Informal approaches are as important as formal approaches and need to be understood if one wants to provide support for software development. Informal approaches solve problems not addressed by formal approaches, so formal and informal approaches complement each other. An example of an informal approach is the use of formal communication channels in software development organizations to deal with dependencies among components of the same subsystem when the developers are co-located [9].

In this paper, we describe an informal field study that analyzes both formal and informal approaches used by a software development team to manage the interdependencies that occur during software development. We classify this work as an informal study since it consists primarily of observations made by the first author during an eight-week internship during the Summer of 2002. The formal

approaches identified here are those legitimately adopted by the organization, such as the software development process; the software development tools used, namely the configuration management (CM) and bug-tracking tools; and other approaches, such as the division of labor, formal meetings, and so on. The informal approaches are the emerging practices adopted by the team to deal with these interdependencies, such as the adoption of conventions; partial check-ins; problem reports (PRs) that cross work boundaries; and the role of e-mail as a coordination mechanism. Our observations build on Grinter's work [9]; we identify several other informal approaches and analyzed the role of formal approaches in the management of interdependencies. The identification, analysis, and support for formal and informal approaches are essential in improving software development efforts. Interdependencies affect the coordination success because they decrease the certainty of a project [13].

2. The MVP Software Development Team

The field study was conducted in cooperation with a team that develops a software application, which for the purposes of this paper we call MVP (All names were changed to preserve anonymity). MVP is a suite of 10 different tools developed at NASA/Ames Research Center. The MVP source code is approximately one million lines of C and C++.

2.1. Team Organization

The MVP team is divided in two groups: developers and V&V staff. Developers are responsible for writing new code, fixing bugs, adding new features, and so on. This group comprises 25 members, 3 of whom are also researchers who write their own code to explore new ideas. The overall experience of these developers ranges from 3 months to more than 25 years. Experience in the MVP group ranges from 2.5 months to 9 years. This group is spread along several offices across two floors in the same building.

V&V members are responsible for testing and reporting identified bugs, keeping a running version of the software for demonstration purposes, and maintaining the documentation (mainly user manuals) of the software. This group comprises 6 members, half located in the V&V Laboratory, and the rest in several offices on the same floor as the laboratory. The V&V Lab and the developers' offices are located in the same building.

2.2. The MVP Software

Each of the MVP's 10 tools uses a specific set of "processes." A process, for the MVP team, is a program that runs with the appropriate run-time options. It is not formally related to the concept of processes in operating systems and/or distributed systems. MVP's processes typically run on distributed Sun workstations and communicate using a TCP/IP socket protocol. Running a MVP tool means running the processes required by this tool with their appropriate run-time options. Processes are used to divide the work among the developers (see section 4.3).

3. Methods

As an intern with the MVP team, the first author was able to make observations and collect information about several aspects of the team. Additional material was collected by reading manuals for the MVP tools, manuals for the software development tools used, formal documents (such as the description of the software development process and the ISO 9001 procedures), training documentation for new developers, problem reports, and so on, as well as talking to colleagues. Some of the team members—the documentation expert, V&V members, testers, process leaders, and process developers—agreed to let the intern shadow them for a few days to better learn about their functions and responsibilities. A representative subset of the MVP group was interviewed. Interviews lasted between 45 to 120 minutes. A total of seven interviews [15] were used to find out about the usage patterns of various tools. The data has been analyzed by using grounded theory [28].

4. Formal Approaches

Formal approaches are those legitimately adopted by the team to support the management of interdependencies. They facilitate the software development effort by improving the coordination of activities. These approaches have long been studied in the software engineering and organizational research literature (e.g., [6, 26]), so we will mention only aspects of these approaches in the context of the MVP team.

4.1. The Software Development Process

The MVP team uses a formal software development process that prescribes the steps needed to be performed by the developers. For example, the following steps must be performed by all developers after finishing the implementation of a change. Initially, they should integrate their code with the main baseline. After that, must test their changes to check if their integrations have inserted bugs in the code. Finally, after checking-in files into the repository, developers must send e-mails to the software development mailing list describing the problem report (PR) associated with the changes, the files that were

changed, and the branch where the check-in will be performed, among other pieces of information.

The MVP software process also prescribes the usage of code reviews before the integration of any change and design reviews for major changes in the software. Code reviews are performed by the manager of each process. Therefore, if a change involves two processes, a developer's code will be reviewed twice: once by each manager. Design reviews are recommended for changes that involve major reorganizations of the source code; their use is decided by the software manager.

4.2. The CM and Bug Tracking Tools

We observed that MVP developers employ mainly two software development tools for *coordinating* their work: a configuration management (CM) system and a bug-tracking system [2, 9, 11]. These tools are integrated so that there is a link between the PRs (in the bug-tracking system) and the respective changes in the source code (in the CM tool). Both tools are provided by one of the leader vendors in the market. Other tools, such as CASE tools, compilers, linkers, debuggers, and source-code editors, are also used.

A CM tool supports the management of source-code dependencies through its embedded building mechanisms, which indicate what parts of the code need to be recompiled when one file is modified. In this case, we use Grinter's classification of dependencies: "Compile-time dependencies occur when a sub-system is being compiled. Build-time dependencies occur when several subsystems or the entire system is being linked. Run-time dependencies occur when the executable is running [9]." According to this classification, CM tools support compile and build-time dependencies. Similarly, a bug-tracking tool, when associated with the CM tool, supports the tracking of changes performed in the source code during the development effort.

Two members of the MVP team play important roles in the usage of these tools: the configuration and release manager and the bug-tracking manager. Both help in the administration of the tools and try to relieve the developers of some of most common tasks (e.g., the CM manager created a command interface on top of the CM tool to make it easier for MVP developers to use). The CM manager provides full-time support for the CM tool, and the bug-tracking manager is also an MVP software developer. Both managers have been receiving training in those tools, and other developers are trained before starting work in the group. Their training includes the software development tools and the MVP software development process.

The MVP team employs several advanced features of the CM tool, such as triggers, "winking in" techniques to reduce compilation time, labeling, and branching strategies. Indeed, the branching strategy employed is one of the most important aspects of a CM tool because it principally affects the work of MVP developers. It is a way of deciding when and why to branch. This strategy affects the task of coordinating parallel changes. According to the nomenclature proposed by Walrad and Strom [31], the following branching strategies are used by the MVP team: (1) branch-by-purpose, in which all bug fixes, enhancements, and other changes in the code are implemented on separated branches; (2) branch-by-project, in which branches are created for some of the development projects; and (3) branch-by-release, in which the code branches upon a decision to release a new version of the product. The branch-by-purpose strategy is employed by MVP developers in their daily work, whereas the other strategies are used only by the CM manager. In other words, the developers themselves create new branches for each new bug fix or enhancement, but branches for projects and releases are created only by the manager.

The branch-by-purpose strategy supports a high-level of parallel development by allowing developers to work on different branches at the same time, thus avoiding problems that exist in other strategies [31]. According to this strategy, each developer is responsible for integrating his or her changes into the main code, which is often called "push integration" [1]. The changes are then available to all other developers. Therefore, if one bug is introduced, other developers will notice it because their work will be disrupted. Indeed, we observed and collected reports of different instances of this situation. A developer who suspects there is a problem introduced by recent changes will contact the author of the changes to check the change, or to provide more information about it.

4.3. Other Approaches: Meetings and Division of Labor

MVP developers employ other formal approaches to manage the interdependencies in the software. For example, the V&V group holds weekly meetings to discuss problems, deadlines, etc. These meetings are also used for official announcements, such as trips, dates of new releases, demonstrations, audits, and so on. Likewise, the entire MVP team (developers and V&V staff) holds bi-weekly "*software pre-design meetings.*" In these meetings, formal announcements are also made, but the most important part of the meeting involves the discussion of new PRs. In this case, the developers each announce their new PRs, describing them through their number and headline. In general, the headline provides enough information about the nature of the PR, but other developers might ask for more details. This is an opportunity for developers to discuss their work, obtain help, and be aware of what is happening in the team. For example, it is not

uncommon after a developer reports a PR that another developer mentions that the problem has already been fixed. PRs that are almost finished might also be announced to warn others about possible "weird" behavior in the tools. Finally, during these meetings the software manager will decide if design reviews are necessary.

The MVP software development team also adopts a clear division of labor based on the processes that compose each MVP tool. Each developer is assigned to one or more processes and tends to specialize in it. There are process leaders and process developers, who mostly work only on a particular process. This is important because it allows the developers to understand the behavior of the process more deeply and become familiar with its structure, therefore helping them to deal with the complexity of the code. Indeed, during the software development activity, managers tend to assign work according to these processes. However, it is not unusual to find developers working in different processes under various circumstances (e.g., before launching a new release, a developer might be assigned to fix bugs in other processes). Developers also work in different processes due to the continuity of the work. Sometimes bugs that seem to be located in a process and therefore are allocated to the developer who works with this process are later discovered to be located in another process. In this case, it is better to let the developers finish the work because so much time was invested in it. Thus, this allows developers to gain a comprehensive view of the whole MVP software.

5. Informal Approaches

Informal approaches are the practices adopted by the MVP team to deal with the interdependencies that occur during the software development process. We call them informal because they emerged naturally in response to the needs of the team and are not taught to new members. The approaches that we identified are discussed below.

5.1. Problem Reports Are Boundary Objects

In our analysis we identified that PRs are used to facilitate the management of interdependencies of developers from different groups and with different roles. In other words, PRs are "boundary objects" in the sense of Star and Griesemer [27]: objects whose common identity is robust enough to support coordination, but whose internal structure, meaning, and consequences emerge from local negotiations between groups. Indeed, PRs are used by end-user liaisons, developers, and testers for different purposes.

Consider the following. When a bug is identified, it is associated with a specific PR. Whoever identified the problem is also responsible for including information about 'how to repeat it' in the PR. This description is used by the developer assigned to fix the bug to specify the circumstances (adaptation data, tools, and their parameters) under which the bug appears. After fixing the bug, this developer must fill a field in the PR that describes how the testing should be performed to properly validate the fix. This field is called 'how to test.' This information is then used by the test manager, who creates test matrices that will be used later by the testers during regression testing. The developer who fixes the bug also indicates in another field of the PR whether the documentation of the tool needs to be updated. Then, the documentation expert uses this information to determine whether the manuals need to be updated based on the changes the PR introduced. Finally, another field in the PR conveys what needs to be checked by the manager when closing it. Therefore, the PR reminds the software manager of the aspects that need to be validated.

In short, the information provided by the PR is used by the developers to manage the several interdependencies in the software being developed. For example, since the user manual of an MVP tool depends on part of that tool's source code, so changes in this source code need to be reflected in the manual. The information about such changes is provided to the documentation expert through one of the fields in the PR.

5.2. Naming Conventions

Developers share repositories containing the source code (the CM tool) and information about changes in this code (the bug-tracking tool). As a result, the team establishes naming conventions that must be followed when dealing with these tools. Conventions are common and accessible rules or arrangements established in the group that act as a means to merge the different perspectives and work styles involved in handling shared objects [14].

An example of a convention is the naming convention used in the creation of branches in the CM tool: it must be based on the PR number recorded in the bug-tracking tool as well as on the developer's name. This allows the relationship that exists between a change and its corresponding PR to be clearly represented, therefore facilitating identification by MVP developers. However, these conventions are not properly supported by these tools, which is a source of complaints by the developers. Indeed, creating and naming branches is a cumbersome task with four or five different tedious steps that could be automated because they follow a naming convention.

5.3. E-mail Conventions

As mentioned before, the MVP software development process prescribes that after checking-in code into the repository, a developer needs to send an e-mail to the software developers' mailing list. However, we found out

that MVP developers perform these activities in the reverse order—they will send e-mail before, not after, the check-in. By doing so, MVP developers allow their colleagues to prepare for the changes. Indeed, developers might even send e-mail to the author of the change asking for a delay of its check-in. We also found out that in this same e-mail developers describe the impact that their changes will have on others' work. A developer who reads these e-mails might walk to the co-worker's office to ask about the changes or, if the change has already been committed, browse the CM and bug-tracking systems to understand them. The following list presents some usual comments sent by MVP developers:

"No one should notice."
"(…) only EDP users will notice any change."
"Will be removing the following [x] *file. No effect on recompiling."*
"Also, if you recompile your views today you will need to start your own [z] *daemon to run with live data."*
"The changes only affect [y]*-mode so you shouldn't notice anything."*
"If you are planning on recompiling your view this evening ([current date]) and running an MVP tool with live [z] *data, you will need to run your own* [z] *daemon."*

Sending e-mail before the check-ins with the description of the impact of the changes is an important convention because it allows other developers to prepare and reflect about the effect of their colleagues' changes in their current work. Because they are aware of some of the interdependencies in the source-code, they might consequently adjust to these changes.

In addition to the flexibility that allows the description of the impact of the changes, e-mail provides asynchronous communication, which requires storage of the messages until their delivery to the recipient. This is used by MVP developers to learn about what changed in the code in a certain timeframe. For example, these e-mails were used by a developer to catch up with the changes that occurred while out of the office. They contained information that allowed the developer to identify changes that did not affect current work, but might affect future work. The following comment from another MVP developer supports this:

"(…) all of the sudden you were working and everything was going great and an e-mail comes through, you look at it, it does not mean a lot, you blow it (…) you keep working and one hour later things were broken. Why is that not working? Oh, that last check-in! You go back to that e-mail: who did this? And maybe you can go talk to that person: 'you broke something' (…)"

The information in the e-mail is also important because it informs (or reminds) developers that they have been engaged in parallel development. Often, developers are unaware of parallel activity because they do not check the version tree that displays information about other developers working on the same file. The information in the e-mail is usually enough to tell the developer whether these changes should be incorporated right away or whether they can wait until just before check-in. In either case, the latest changes must be "merged back" into the developer's version of the file. In general, if one file has been checked-in several times and a developer has the same file checked-out, he or she "merges back" the changes indicated in the e-mail to avoid working with an outdated file.

The asynchronous nature of e-mail could be problematic because developers might miss important notifications about changes. However, during the field work, we did not notice any such problems. Furthermore, sending e-mail before a check-in is also used by other developers to support expertise identification and as a learning mechanism. Developers associate the author of the change with the "process" where the changes are being performed. In other words, MVP developers assume that if one developer constantly and repeatedly performs check-in in a specific process, it is very likely that the developer is an expert on that process. Therefore, another developer needing help with that process will look to that developer for help:

"[talking about a bug in a process that he is not expert] *(…) I don't understand why this behaves the way it does. But, most of these PR's seem to have John's name on it. So you go around to see John. So by just by reading the* [PR] *headline of who does what, you kind of get the feeling of who's working on what (…).So they* [e-mails] *tend to be helpful in that aspect as well. If you've been around for ten years, you don't care, you already know that* [who works with what], *but if you've been here for two years that stuff can really make difference (…)"*

In addition, the simple fact that developers read the e-mails sent by other developers to check for the impact of others' changes facilitates learning about the MVP software. Interestingly, the two developers who reported these aspects of e-mail were relative novices at MVP, with 2 years and 2.5 months experience there.

5.4. Holding onto Check-ins

As mentioned, MVP developers add to the e-mail the description of the impact of their changes in other developers' code. The two most common types of impact statements are changes in run-time parameters of a process and the need to recompile parts or the whole source

code[1]. The former case is very important because other developers might be running the process that will be changed. The latter case is described because when a file is modified, it, as well as the other files that depend on it, will be recompiled, and this recompilation process is time-consuming—up to 45 minutes. Developers are aware of the delay they might cause to others; therefore, they hold check-ins until the evening. According to one of the developers:

> *"(…) and the other thing that you find is that when people also know that if they are going to check-in a file they will do in the later afternoon … you're gonna do a check-in and this is gonna cause anybody who recompiles that day have to watch their computer for 45 minutes (…) and most of the time, you're gonna see this coming at 2 or 3 in the afternoon, you don't see folks (….) you don't see people doing* [file 1] *or* [file 2] *checking-in at 8 in the morning, because everybody all day is gonna sit and recompile."*

Holding onto check-ins is an informal approach adopted by the MVP software development team to minimize the problems caused by the interdependencies that exist on the source code. However, this is possible only because MVP developers are aware of the existing interdependencies.

5.5. Engagement in Parallel Development: Partial Check-ins and "Speeding Up" the Process

We also noted that MVP developers engage very often in parallel development. This happens when more than one developer has the same file checked-out. Conflicts might occur when one of these developers checks-in this file back into the repository because the other developer's version will then be outdated, and any changes that developer makes will potentially be inappropriate. To update the version, the developer needs to merge the other's changes back in his or her code. This operation is called by the developers "back merging," and in CM terminology is named "synchronization of workspaces." Due to the need to perform these back merges, a new dependency between artifacts is created during parallel development. This dependency occurs between any version of a file that has not yet been checked-in and the new version of this same file created after the check-in (i.e., the current version of a file checked-out by a developer is now dependent on the new version checked-in into the repository because the former needs to incorporate the changes of the latter before being checked-in). This is another example of dependency in software development.

Conflicting changes are more likely to occur in files that are accessed by several developers at the same time. For example, in MVP software, some files are used to define programming language structures that are used all over the code. Different developers often change these files, which means that they have a high degree of parallel development. These files are especially important because there is a significant correlation between them and the number of defects reported [20]. MVP developers reported that they do not avoid parallel development in these files because conflicts are infrequent and not likely to occur. But, without access to the CM tool, it was not possible to statistically test this claim. MVP developers accepted parallel development because it was necessary to achieve high productivity. However, we identified that they adopted a strategy to deal with files with a high degree of parallel development. To minimize the possibility of conflicts, developers would perform "partial check-ins," which consists of checking-in some of the files back into the repository, even when the developers have not yet finished all their changes. This strategy decreases the number of dependencies that occur, and consequently reduces the number of necessary back merges. Note that partial check-ins are variations of the formal software development process, which establishes that check-ins will be performed only when the changes in all files are finished.

Finally, according to Grinter [9], software developers might rush to finish their work when they engage in parallel development because they want to avoid merging. We identified that developers will rush only when they are testing their changes right before check-in. As one developer plainly pointed out: "This is a race!" According to the software development process, this testing is necessary to guarantee that the changes will not introduce bugs into the system. We observed that this testing is very informal. For example, developers will sit in the V&V Laboratory and compare the current version of the MVP with the one with changes. In short, MVP developers do not use regression testing at this moment. That will be used by the V&V staff before creating a new release of the software. This means that techniques that minimize the number of test cases necessary to validate the changes in the software (e.g., [23]) cannot be used by MVP developers to determine whether the tests they need to run can be impacted by changes that another developer makes. These techniques can be used only by the V&V staff.

Although we observed that some check-ins introduced errors, we do not have evidence that these errors were introduced due to this "racing." Similar to partial check-ins, "speeding up" the process is employed by the MVP developers to avoid the additional work necessary to deal with the extra-dependencies that occur during parallel development.

---

[1] The CM tool used by the MVP team allows developers to choose if they want to incorporate others' changes, meaning that they are able to decide if they want to recompile the code or not.

6. Computational Support for Informal Approaches

Figure 1 summarizes the formal and informal approaches used by the MVP team to manage the interdependencies that occur during their software development activities. As mentioned before, formal and informal approaches complement each other, so problems not solved by the formal approaches might be solved by the informal ones. For example, none of the formal approaches used by the MVP team addresses the issue of how to manage the crossing-boundaries dependencies that occur when a change is committed into the repository. This problem is solved by the MVP team by adopting a particular PR structure that provides information for developers with different roles (see section 5.1).

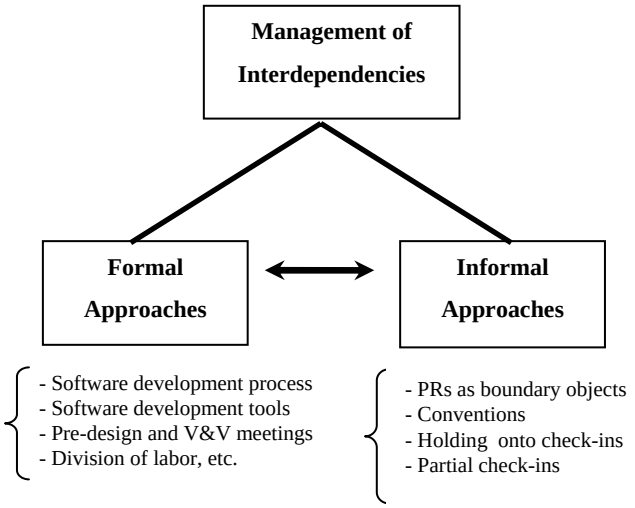

Figure 1: Formal and Informal Approaches Adopted by the MVP Software Development Team

The tools used by the MVP team assist some of the informal approaches. For example, the CM tool allows software developers to perform partial check-ins. In contrast, due to the lack of tool support, developers need to rush to finish their work when they are testing their changes. In this section, we discuss the existence (or lack) of support for informal approaches in more detail. In addition, we discuss implications for software engineering research when there is a lack of support.

6.1. Problem Reports as Boundary Objects

Bug-tracking tools are flexible enough to allow their managers to define the fields that will compose a PR. In addition, these tools allow a manager to specify a simple workflow describing *when* each one of these fields needs to be filled in [12]. By doing that, they allow the creation of PRs with fields that contain information that is useful to developers who are members of different groups. In the MVP team, the information in these fields describes how each developer's work is going to be affected by the PR. This means that these tools allow PRs to be defined and used as coordination mechanisms to manage interdependencies during software development.

6.2. Support for Naming Conventions

Following conventions for dealing with shared objects (or repositories) implies additional effort; hence, technical support often is needed [14]. As mentioned before, MVP developers follow a naming convention in which the name of the branches in the CM tool should be based on the PR number recorded in the bug-tracking tool. MVP developers have complained that the task of creating branches is very cumbersome, with four or five different tedious steps to be performed. Because this task is based on a convention, it could be automated. Unfortunately, the current integration between the CM and the bug-tracking tool does not support that. That is a major source of complaints repeatedly reported by the MVP software developers during the interviews.

6.3. Support for E-mail Conventions

NASA requires ISO 9001 certification for all software development efforts, which means that all changes in the software must be documented, reviewed, and formally authorized before the changes are integrated in the code. In other words, developers need to be accountable for their work. The MVP team chose to use e-mail as a formal communication channel in the organization, as clearly mentioned in the software development process. Indeed, some of the tasks (such as requesting and answering code reviews) were performed by using e-mail. These tasks require the use of software development tools such as source-code editors, CM tools, and so on. Unfortunately, e-mail is not integrated with these tools, which means that developers need to move back and forth between e-mail and the other tools in order to get their work done. Integration of e-mail with software development technology seems easy to implement; it is also very promising because more and more software development organizations are seeking certifications such as ISO 9001 and CMM (Capability Maturity Model). This aspect was identified during the field work and later corroborated by MVP software developers during the interviews. In addition, e-mail messages exchanged among developers are also used to identify expertise in parts of the source code, as well as a history mechanism to identify changes that happened in the past. Again, this information could and should be properly organized and indexed in order to facilitate these activities.

6.4. Holding onto Check-ins

The informal approach of holding onto check-ins is used to avoid disrupting others' work. The support for this task provided by CM tools is appropriate because these tools allow a developer to check files in or out and merge different versions of them at any time. However, this approach is useful only if the developer who is going to check-in some code is aware that his or her work will cause the recompilation of other files. This suggests that software visualization tools (e.g., [4]) that use existing information from the CM tool could be used to support the identification of these files by novice developers who are not aware of the interdependencies in the source code.

6.5. Partial Check-ins

A check-in is called "partial" by the MVP developers when it is performed without a code review to avoid several "back merges" due to the file being changed by several other developers at the same time. CM tools support partial check-ins because they usually do not impose constraints about when check-ins might be performed, allowing one to check-in code into the repository at any time. However, the current trend of integrating CM tools with software process technology [5] might disrupt that. We recognize this integration is essential because it allows the efficient automation of repetitive tasks (such as building a software release) [12]. Nevertheless, the enforcement of the process that usually goes along with this integration must be managed, because it has long been recognized as problematic [29]. CM tools must be flexible enough to allow software developers to use workarounds that deviate from the process in order to properly deal with the problems that they face. One example of such workarounds is the partial check-in. Another approach is to update the software development process to reflect the need for partial check-ins, and consequently legitimate them. In this case, similar to holding check-ins, the information already present in the CM tool could be used by software visualization tools [4] to allow novice developers to identify files with a high degree of parallel development that need to be partially checked-in.

6.6. Speeding Up the Process

MVP developers rush their activities during the development process to minimize the number of dependencies between their code and recently committed changes in the repository (section 5.5). Current CM and bug-tracking tools create the need to speed up because they shield a developer's workspace from other developers' workspaces to support parallel development. Although it is desirable to isolate one developer's work from others, it does not allow developers to coordinate their check-ins, and hence avoid the need to re-do their work. To the best of our knowledge, no existing software engineering tool solves this problem. However, a promising approach recently emerged with tools that attempt to break the isolation of CM workspaces (e.g., [24] and [17]). These tools achieve that by distributing the CM commands happening in a developer's workspace to other selected workspaces. These tools focus on the actions of the developers (conveyed as CM commands) because they want to avoid conflicts between the files that two or more developers have checked-out. In addition, we argue that these tools need to provide information about the "status" of other developers' work. By doing that, they allow a developer to identify who is about to check-in code into the repository and, therefore, to coordinate their work, so that a developer does not need to rush. We believe that this can be achieved by extending these tools to collect information from sources other than the CM tool, such as e-mail, the bug-tracking tool, the software process specification, and so on.

7. Discussion

As mentioned before, a formal process description can never completely represent all variations that might occur in a software development effort [8]. Therefore, as the data have suggested, informal approaches need to be adopted to complement the formal approaches to properly support the management of the interdependencies that occur in the software development process. However, to properly support cooperative software development, we need to unveil these informal approaches and provide computational support for them to minimize errors and improve their performance. One of the reasons these informal approaches are important is the high level of parallel development that occurs in large-scale collaborative efforts [20]. Indeed, the engagement in parallel development identified in this field study helps to substantiate the results of Perry et al. [20] that describe high levels of parallel development, but contrasts with the groups studied by Grinter [9, 11], in which developers avoided this situation. Technical improvements in merging techniques from 1995 to 2002 [2] might be the cause of divergence from Grinter's earlier observations. Grinter, however, does not clearly describe the branching strategy used by the team studied, whereas the MVP team adopted the "branch-by-purpose" strategy. According to Walrad and Strom [31] this "strategy supports a high level of parallel development by allowing developers to work on different branches at the same time. Therefore, this might be another explanation for the difference between the two groups. Finally, an organization's structural properties (e.g., reward systems, policies, norms, and so on) are other factors that influence the adoption and use of col-

laborative tools [18]. The two organizations studied are different, hence they are very likely to have different structural properties, which might explain the different levels of engagement in parallel development.

Meanwhile, this field study supports Grinter's [9] finding that during parallel development developers will rush to finish their changes. However, while the developers studied by Grinter will speed up because they want to avoid the complexity of merging, MVP developers rush because they do not know when another developer might check-in some code that will lead them to another set of tests. In both studies, developers describe their dilemma: they want to produce high-quality code, but they also want to finish their changes fast.

The MVP team needs to perform extra work to successfully manage the interdependencies in the software. This extra work is a form of articulation work necessary to coordinate, negotiate, mesh, and schedule their activities [25]. It is different from recomposition work [10], which is the coordination required to assemble software development artifacts from their parts, because recomposition work focuses on choosing the right components to create a software artifact due to source-code dependencies, whereas this extra work focuses on the management of all dependencies that exist in a software development effort.

Finally, in this informal field study we identified another approach used by software developers to identify experts. Whereas McDonald and Ackerman [16] describe the usage of change history data (equivalent to PRs in the MVP team), novice developers in the MVP team use the broadcasted e-mail messages prescribed by the software development process. The importance of finding experts for problem-solving in any organization and the complexity of the MVP code suggest that the operation of sending e-mail before a check-in is essential.

8. Conclusion and Final Remarks

This paper reports the findings of an informal field study conducted at the NASA/Ames Research Center during the course of an eight-week internship with a software development. The results of this field study describe the formal and informal practices adopted by team members to manage the interdependencies that occur during software development. Formal approaches are those legitimated by the organization; the informal ones are those that emerge naturally due to the needs of the developers. Examples of formal approaches adopted by the team are the software development process, some software development tools, design meetings, and a clear division of labor. The informal approaches that we identified are partial check-ins, problem reports that cross work boundaries, holding onto check-ins, e-mail and naming conventions, and the action of speeding up the processes.

In this work, we also indicate current and nonexisting computational support to the informal approaches. Indeed, partial check-ins, problem reports that cross work boundaries, and holding onto check-ins are work practices currently supported by CM and bug-tracking tools. E-mail and naming conventions and the action of speeding up the processes are adopted by MVP developers due to the lack of tool support. We believe that these interesting research areas should be further investigated. Pointing out these areas is an important contribution of this paper.

Finally, we are planning a future study in a different organization. We seek to identify similarities and differences in the formal and informal approaches that we identified here and to learn how the ones that we identified are used in a different context.

Acknowledgments

The authors thank CAPES (grant BEX 1312/99-5) and NASA/Ames for financial support. This effort was also sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Funding also was provided by the National Science Foundation under grant numbers 0205724 and 0083099. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force Laboratory, or the U.S. Government.

9. References

[1] Appleton, B., Berczuk, S., et al., "Streamed Lines: Branching Patterns for Parallel Software Development," Proceedings of Pattern Languages of Programs (PLoP'98), Washington University Technical Report #WUCS-98-25, 1998.

[2] Conradi, R., and Westfechtel, B., "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, pp. 232-282, 1998.

[3] Curtis, B., Krasner, H., et al., "A Field study of the Software Design Process for Large Systems," *Communications of the ACM*, vol. 31, pp. 1268-1287, 1988.

[4] Eick, S. G., Graves, T. L., et al., "Visualizing Software Changes," *Software Engineering*, vol. 28, pp. 396-412, 2002.

[5] Estublier, J., "Software Configuration Management: A Roadmap," Future of Software Engineering, pp. 279-289, Limerick, Ireland, 2001.

[6] Finkelstein, A., Kramer, J., et al., *Software Process Modeling and Technology*: Wiley, 1994.

[7] Gamma, E., Helm, R., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addi-

son-Wesley, 1995.
[8] Gerson, E. M., and Star, S. L., "Analyzing Due Process in the Workplace," *ACM Transactions on Office Information Systems*, vol. 4, pp. 257-270, 1986.
[9] Grinter, R., "Supporting Articulation Work Using Configuration Management Systems," *Computer Supported Cooperative Work*, vol. 5, pp. 447-465, 1996.
[10] Grinter, R. E., "Recomposition: Putting It All Back Together Again," Conference on Computer Supported Cooperative Work (CSCW'98), pp. 393-402, 1998.
[11] Grinter, R. E., "Using a Configuration Management Tool to Coordinate Software Development," Conference on Organizational Computing Systems, pp. 168-177, 1995.
[12] Grinter, R. E., "Workflow Systems: Occasions for Success and Failure," *Computer Supported Cooperative Work*, vol. 9, pp. 189-214, 2000.
[13] Kraut, R. E., and Streeter, L. A., "Coordination in Software Development," *Communications of the ACM*, vol. 38, pp. 69-81, 1995.
[14] Mark, G., Fuchs, L., et al., "Supporting Groupware Conventions through Contextual Awareness," European Conference on Computer-Supported Cooperative Work (ECSCW '97), pp. 253-268, Lancaster, England, 1997.
[15] McCracken, G., *The Long Interview*: Thousand Oaks, CA: SAGE Publications, 1988.
[16] McDonald, D., and Ackerman, M., "Just Talk to Me: A Field Study of Expertise Location," Conference on Computer Supported Cooperative Work, pp. 315-324, 1998.
[17] O'Reilly, C., Morrow, P., et al., "Improving Conflict Detection in Optimistic Concurrency Control Models," 11th International Workshop on Software Configuration Management (SCM-11), Portland, Oregon, 2003.
[18] Orlikowski, W., "Learning from Notes: Organizational Issues in Groupware Implementation," *The Information Society*, vol. 9, 1993.
[19] Parnas, D. L., "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, pp. 1053-1058, 1972.
[20] Perry, D. E., and, Siy, H. P., et al., "Parallel Changes in Large-Scale Software Development: An Observational Case Study," *ACM Transactions on Software Engineering and Methodology*, vol. 10, pp. 308-337, 2001.
[21] Perry, D. E., Staudenmayer, N. A., et al., "People, Organizations, and Process Improvement," *IEEE Software*, vol. 11, pp. 36-45, 1994.
[22] Podgurski, A., and Clarke, L. A., "The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance," Symposium on Software Testing, Analysis, and Verification, pp. 168-178, 1989.
[23] Rothermel, G. and Harrold, M. J., "A Safe, Efficient Regression Testing Selection Technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 173-210, 1997.
[24] Sarma, A., Noroozi, Z., et al., "Palantír: Raising Awareness among Configuration Management Workspaces," Twenty-fifth International Conference on Software Engineering, pp. 444-453, Portland, Oregon, 2003.
[25] Schmidt, K., and Bannon, L., "Taking CSCW Seriously: Supporting Articulation Work," *Journal of Computer Supported Cooperative Work*, vol. 1, pp. 7-40, 1992.
[26] Shull, F., Carver, J., et al., "An Empirical Methodology for Introducing Software Processes," Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 288-296, Vienna, Austria, 2001.
[27] Star, S. L., and Griesemer, J. R., "Institutional Ecology, Translations and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology," *Social Studies of Science*, vol. 19, pp. 387-420, 1989.
[28] Strauss, A., and Corbin, J., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, Thousand Oaks, CA: SAGE publications, 1998.
[29] Suchman, L., *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge: Cambridge University Press, 1987.
[30] Vessey, I., and Sravanapudi, A. P., "CASE Tools as Collaborative Support Technologies," *Communications of the ACM*, vol. 38, pp. 83-95, 1995.
[31] Walrad, C., and Strom, D., "The Importance of Branching Models in SCM," *IEEE Computer*, vol. 35, pp. 31-38, 2002.