Continuous Coordination: A New Paradigm for Collaborative Software Engineering Tools

André van der Hoek, David Redmiles, Paul Dourish
Anita Sarma, Roberto Silva Filho, Cleidson de Souza
Department of Informatics
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{andre, redmiles, jpd, asarma, rsilvafi, cdesouza}@ics.uci.edu

Abstract

Collaborative software engineering tools that have been developed and used to date exhibit a fundamental paradox: they are meant to support the collaborative activity of software development, but cause individuals and groups to work independently from one another. The underlying issue is that existing tools discretize time and tasks in concrete but isolated process steps. This approach is fundamentally flawed in assuming that human activity can be codified and that periodic resynchronization of tasks is an easy step. We propose a new approach to supporting collaborative work called continuous coordination. The underlying principle is that humans must not and cannot have their method of collaboration dictated, but should be supported flexibly with both the tools and the information to coordinate themselves and collaborate in their activities as they see fit. In this paper, we define the concept of continuous collaboration, introduce our work to date in building some example tools that support the continuous coordination paradigm, and set out a further research agenda to be pursued.

1. Introduction

Collaboration is at the heart of software development. Most software is developed by a group of people rather than an individual. This means that software engineering environments and processes must support the coordination of the activities that individuals carry out within the overall objectives of the group at large. Within the software engineering community, the response has been a flurry of research and development that has resulted in the availability of a host of formal software process languages and environments. These languages and environments aim to help users in choosing their tasks, obtaining prerequisite input from a set of relevant people and tools, performing their tasks, and sending their output to another (sometimes overlapping) set of relevant people and tools. While this has certainly helped in advancing the ability of individuals to collaborate in groups, these approaches are built on a fundamental paradox: to collaborate, individuals work completely independently from each other and are isolated by the environment that supports their day-to-day activities.

Within the computer-supported cooperative work community, the response has been virtually the opposite rather than constraining and guiding a user in their tasks, the focus is on informally raising awareness by informing users of ongoing, parallel activities so they can interpret this information and self-coordinate amongst each other. While this has lead to novel tools and approaches, there is an issue of scalability and cognitive overload: users can only absorb and meaningfully interpret limited amounts of typically contextualized information.

In this paper, we introduce and explore an alternative approach: continuous coordination. Continuous coordination blends the best aspects of the more formal, process-oriented approach with those of the more informal, awareness-based approach. In doing so, continuous coordination blends processes to guide users in their day-to-day high-level activities with extensive information sharing and presentation to inform users of relevant, parallel ongoing activities. Through this blending, users become aware of the context in which they perform their work, can interpret their context, and take action accordingly. This allows users to self-coordinate within the overall process to avoid situations in which their activities threaten to obstruct or interfere with activities of others, or simply to better organize and order respective tasks.

2. Motivating Example

To better understand why, when, and how software developers coordinate their work, we conducted an eight-week field study of existing software development practices at NASA/Ames Research Center [1,2]. In particular, we studied a software team developing a suite of tools to help air traffic controllers manage the increasingly complex air traffic flows at large airports. The team is composed of 25 co-located developers, who design, test, document, and maintain the tools. Like most professional software teams, they make use of an advanced configuration management system, which they use to periodically check out and check in chunks of code, as well as to coordinate their parallel work. However, the configuration management system tells only part of the story. Our field study produced two important observations:

- While having at their disposal a state-of-the-art configuration management system to coordinate their activities, developers mostly relied on an informal mechanism, e-mail, to inform each other about those activities. Before a developer checks in changes, for instance, it is customary to send an e-mail notifying the whole group of not just the imminent action, but also of the effect it may have on the work of everybody else.
- While developers stated in our interviews that the combination of configuration management with e-mail works fine for them, in reality we observed that this is not the case. For example, when they get closer to completing their changes, often they rush to be the first to check in to avoid having to be the person who has to merge and/or retest. As another example, they often do not wait until they have finished their work, but instead try to minimize the possibility of conflicting changes by checking in partially completed work.

These examples highlight a mismatch between the collaboration model supported by the configuration management system and the actual collaboration needs of the developers. While the technology embodies a model of collaborative work designed to ensure team progress with a minimum of coordination problems, in practice we see that these formal mechanisms are accompanied by a set of less formal communicative practices which help to set the formal work in context. In this particular case, the isolation amongst the developers introduced by the CM system is offset via primitive but somewhat disciplined use of e-mail.

Our study is not unique in making these observations. Time and again it has been demonstrated that software engineering tools fail in their attempts to codify human activity (e.g., [3, 4, 5]). They limit the dimensions of human activity and creativity, and therefore are unsuccessful in providing adequate support for effective and flexible collaboration. The resulting problems are dramatic: much time and effort is wasted in resolving conflicting changes; salient faults are introduced as a result of parallel but undetected incompatible changes; and the team as a whole has little intellectual and conceptual integrity. Overall, the software development process remains ineffective and instills a false sense of security in its ability to manage the collaborative effort.

3. Formal Coordination

Many software engineering tools that support coordination and collaboration rely on a formal, process-based approach [6]. A process model, either implicitly or explicitly defined by the tool, splits work into multiple, independent tasks that are periodically resynchronized. This approach, illustrated in row 1 of Table 1, can be characterized as inherently group-centric: it makes the group as a whole the important entity by providing a scalable, predictable, and dependable solution that promotes tight-controlled coordination and insulates different activities from each other. The canonical example is a configuration management system: by checking out artifacts a developer is insulated from other activities and by checking in any modified artifacts the developer resynchronizes their work with the work of the group.

The formal, process-based approach, however, suffers from two significant problems that make it a less-than-effective solution when it comes to coordination and collaboration:

1. Formal processes can describe only part of the activity of software development (or any collaborative task). No matter how formal and well-defined a process may seem, there is always a set of informal practices by which individuals monitor and maintain the process, keep it on track, recognize opportunities for action and the necessity for intervention or deviation. In other words, no process description will or can ever be complete (e.g., [7]).
2. Even when a process description attains a relatively high degree of detail and accuracy, the periodic resynchronization of activities remains a difficult and error-prone task. In fact, it has been shown that when more parties are involved, more conflicts arise and more faults are introduced in the software at hand (e.g., [8]).

There are solid theoretical foundations to back up these observations (e.g., [9, 10]). What the theory indicates is that the empirical phenomena observed are not simply signs of poorly-designed processes or badly-specified tools. Rather, these problems are inherent in any tool that relies upon a formal encoding of collaborative work. Any formal process is inevitably surrounded by a set of informal practices by which the formal conditions are negotiated and evaluated.

4. Informal Coordination

The notion of awareness, as an informal, passively-gathered understanding of the ongoing activities of others, has become a central element of Computer-Supported Cooperative Work (CSCW) research. Through a range of workplace studies, CSCW researchers have begun to recognize the central role played by awareness in collaborative systems [11, 12]. Awareness is an informal understanding of the activity of others that provides a context for monitoring and assessing group and individual activity (such as the mutual awareness of activities that arises in shared physical environments, where we can see and hear each other and "keep an eye out" for interesting or consequential events). Following from these observations, CSCW tool developers began to investigate ways to provide continual visibility (awareness) of concurrent actions in hopes of stimulating its users to self-coordinate.

This approach, illustrated in row 2 of Table 1, can be characterized as inherently user-centric: it places the user first in providing them with a flexible mechanism that promotes intellectual and conceptual integrity and allows users to place their own work in the context of others' activities. The canonical example is the multi-user editor: by continuously displaying the ongoing activities of others, users typically self-coordinate by avoiding areas of the document in which others are currently working.

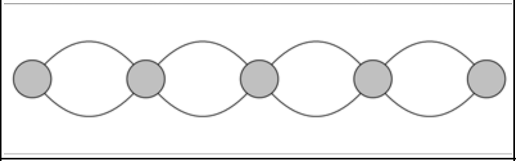
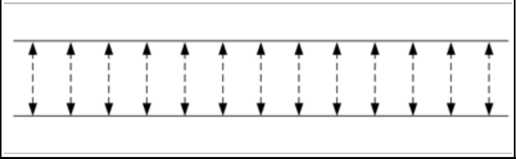
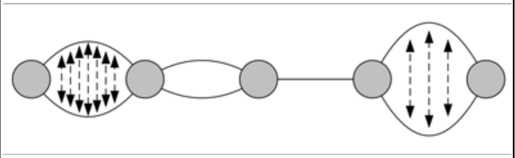
As with the formal, process-based approach, the informal, awareness-based approach suffers from a significant problem that makes it a less-than-effective solution when it comes to coordination and collaboration. In particular, implementations of awareness-based approaches scale poorly; they are largely of value for small groups only. This is primarily caused by two factors:

1. Users have limits on the amount of cognitive information they can process. Especially in complex situations, the amount of "awareness information" generated by a system can be so large that the net effect is that the user ignores all information. Human intermediation is a critical step in this approach, and care must be taken not to cognitively overload users.
2. The emergence of awareness-based approaches as a reaction to the strong restrictions imposed by workflow and process-based technologies means that, in most CSCW research, the approaches have largely been seen as irremediably opposed. As a result, most CSCW tools tend to abandon any form of process altogether and, by purposely only sharing information, leave all coordination tasks to the user.

While some have proposed mechanisms for "asynchronous awareness," which can more easily support large-group collaboration (e.g., [13]), the conventional wisdom is that awareness technologies work well for small groups, but break down for large groups.

5. Continuous Coordination

The formal and informal approaches have thus far always been treated as opposites. Developers have either looked towards formal processes or informal awareness to support coordination. Our research moves beyond this long-standing dichotomy and proposes an integrated approach to supporting collaborative work that combines formal and informal coordination to provide both the tools and the information for users to self-coordinate. The result, which we term *continuous coordination*, is shown

	Conceptual Visualization	*Strengths*	*Weaknesses*
Formal process-based coordination		Scalable; Control; Insulation from other activities; Group-centric	Resynchronization problems; Insulation becomes isolation
Informal, awareness-based coordination		Flexible; Promotes synergy; Raises awareness; User-centric	Not scalable; Requires extensive human intermediation
Continuous coordination		*Expected to be the strengths of both formal and informal coordination*	*To be discovered by future research*

in row 3 of Table 1. Continuous coordination aims to combine the strengths of the formal and informal approaches while overcoming the current shortcomings of either. In particular, it retains the checkpoints and measures of the formal approach to coordination, but provides developers with a view of each others' relevant activities between the formal checkpoints. In doing so, it provides them with ways to understand the potential relationships between their own work and the work of their colleagues. This is not a way to step outside the bounds of formal coordination—rather, it allows developers to better judge both the timing and the impact of formal coordination actions. Neither is it "just a better way" of exception handling—rather, we consider the occurrence of conflicts and other hindrances a normal part of any process and believe that any approach must integrally address them in a combined formal and informal way.

Our goal in promoting continuous coordination is not to create radically new ways of working, but rather to provide more effective technical support for the existing balance between formal and informal approaches. The mechanisms we describe—the need to be able to assess and manage the formal coordination—are already aspects of software development practice. Consider the field study presented in Section 2. Developers recognized the need for formal underpinnings of their effort (the configuration management system) but at the same time realized they also needed informal channels of communication to be more aware of each others' activities (the e-mails). In effect, they are attempting to create a continuous coordination approach, but are forced to do so by combining tools that are not necessarily fully prepared to support that approach. For instance, it is only an accidental side effect of using e-mail that results in the developers being aware of who has expertise in which area. As another example, they wish to be able to check in partially completed tasks to share with specific other developers. The configuration management system, however, only supports them in checking in changes to the central repository and cannot distinguish a "partial check in" directed to a specific person from a "complete check in" directed to the group as a whole. The current set of tools, thus, is neither good at providing the information nor the functionality needed for effective collaboration.

The software engineering community must find new ways of constructing software engineering tools such that they integrally support continuous coordination. Formalisms, including process mechanisms, always will be a necessary part of certain kinds of collaborative efforts, but their effectiveness is exploited best when they provide high-level guidance and support. At lower levels, a process must not try to specify and automate all small steps and all activities; rather it must be flexible and augmented with other mechanisms rooted in the informal, awareness-based approaches. The underlying principle of continuous coordination is that humans must not and cannot have their method of collaboration dictated to them, but should be supported flexibly with both the tools and the information to self-coordinate and collaborate in their activities as they see fit.

6. Our Current Work

Currently, we are involved in two different strands of work in our pursuit of understanding continuous coordination. First, we continue to perform empirical studies of software developers and their mechanisms of collaboration and coordination. Second, we have started construction of software engineering tools in support of continuous coordination. As our empirical studies are ongoing, we report on the second aspect of our work here and describe YANCEES, a highly versatile event notification server that can be used in the construction of novel software engineering tools that support continuous coordination, and Palantír, an example tool that embraces the principle of continuous coordination in enhancing existing configuration management solutions with an informal, awareness-based mechanism.

6.1. YANCEES

Notification servers are brokers for system events, generally following a publisher-subscriber pattern [14]. Software processes that produce events publish their events to a notification server. Software processes that perform processing on events subscribe to events of interest through the notification server. Figure 1 shows a general architecture that supports the notion of awareness and, in general, the continuous coordination concept. End users (e.g., designers, programmers, testers, and others) use software tools (e.g., design environments / IDEs, versioning, email, chat, and others). The usage creates events that are published to an event notification server. Visualizations that keep the same end users aware of all activities are software processes, which are consumers of the events.

The above description makes it seem straightforward that notification servers provide an infrastructure for keeping users aware of events of interests. However, some issues arise in practice: generalization versus specialization of notification servers and services; weak support for customization; and poor support for extensibility.

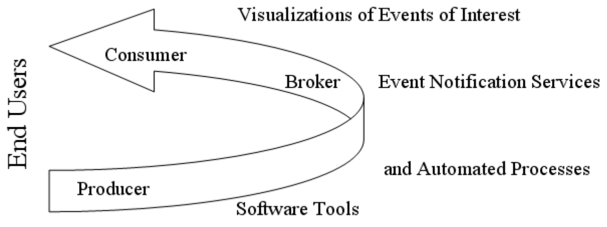


Figure 1. Simplified Architecture Supporting Awareness.

Indeed, a broad spectrum of research and commercial event notification servers are available nowadays. At one extreme, "one-size-fits-all" approaches, such as adopted by CORBA Notification Service [15] or READY [16], strives to address new applications requirements by providing a very comprehensive set of features, able to support a broad set of applications. At the other extreme, specialized notification servers tailored to application-specific requirements provide novel but specific functionalities. Examples of such specialized systems include Khronica [17] and CASSIUS [18] which are specially designed to support groupware and awareness applications; or even Yeast [19] and GEM [20] which are specialized in advanced event processing for local networks applications, and distributed applications monitoring respectively. Finally, servers such as Siena [21] and Elvin [22], even though designed with special domains in mind, strive for a balance between specificity and expressiveness of the subscription language and event model they support.

Therefore, in the development of event-based collaborative software applications, developers face the dilemma of specialization versus generalization: to use a generalized infrastructure, that can support and integrate different applications, but may not provide all the necessary functionality for specific application domains; or to use one event-based infrastructure for each application domain, having "the right tool for the right problem", but losing the uniformity and integration of a single solution.

Another problem of the currently available event-based infrastructures is the weak support for selection and customization of the services to be provided, which is important for applications that run on resource-limited devices such as the ones common to mobile applications.

Moreover, current event-based infrastructures lack mechanisms to support the easy extensibility of its functionality. The only extension mechanism is usually the (understanding and) change of their source code, or the implementation of the service by the client. Additionally, due to restrictions in their event or subscription models, the addition of new functionality may constitute a very difficult task [23].

This work was motivated by the problems discussed above, faced when adapting current notification servers to different collaborative software development scenarios. In an organization, there is a clear need of a single server model that can be adapted and customized to different applications. Moreover, due to the constant evolution of the applications and tools used in these environments, the event-based infrastructure must provide ways to add new features, when necessary, to evolve with the organization's needs.

In order to support this spectrum of requirements, along with the flexibility to select the subset of features needed by each application domain, our event notification architecture had to be configurable and extensible. Such flexibility was one of the main challenges in our design. In short, the architecture needs to:

- Support different requirements associated to the models of the design framework, especially the event, subscription, notification, and resource and protocol models.
- Provide extensibility through mechanisms that allow a programmer to define and implement new capabilities to the models defined above.
- Support different configurations, sets of services (or features) that may work together to provide the functionality necessary for the application domains.
- Permit the distribution of components and services among the publishers and subscribers to comply with device limitations, prevent performance bottlenecks, and implement services that require some degree of distribution, such as mobility.

The YANCEES framework was designed to provide different extension points around a common publish/subscribe core. The extensibility and configurability of the system is achieved by the use of the following strategies:

- Extensible languages to each design dimension
- Dynamic allocated plug-ins
- Parsers that combine and allocate plug-ins according to the extensible languages syntax.
- Input/output filters and shared services as auxiliary elements in the implementation of new extensions
- An architecture configuration manager to statically or dynamically load configurations of plug-ins, services, and filters to each application domain
- A central publish/subscribe core providing basic content-based filtering

The main components and interfaces of the architecture are presented in Figure 2. The static components and APIs are drawn in gray, whereas dynamic allocated components are depicted as dashed line boxes.

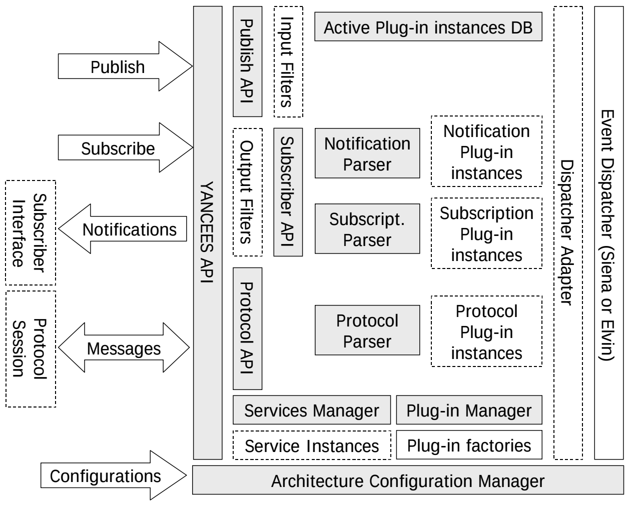


Figure 2. General framework static and dynamic components.

A prototype of the YANCEES framework was implemented using Java 1.4 and the Java API for XML Processing (JAXP) v1.2.3, which supports XMLSchema [24]. The XMLSchema provides inheritance and extensibility mechanisms. When used with the Java DOM parser, an XML document defined according to an XMLSchema can have its syntax automatically validated. This resource is used in our implementation to guarantee the correct form of the messages. The event dispatcher component used in the implementation was Siena 1.4.3. Elvin 4.x was also used in the tests. Both support content-based subscriptions and federation of servers, and both represent events as attribute/value pairs. In our current prototype implementation, two component distribution configurations are possible: (1) the execution of all components on the client stub; or (2) the execution of all components on the server-side. In the former case the communication between client stub and server side (Siena or Elvin) is performed by using their native protocols; in the latter case, the communication between client stubs and the YANCEES server is performed by using Java RMI.

Building upon popular models of event notifications, but using an extensible, sophisticated architecture, YANCEES is able to support many kinds of coordination and collaboration services. YANCEES provides an infrastructure upon which awareness can become part of a suite of software tools.

6.2. Palantír

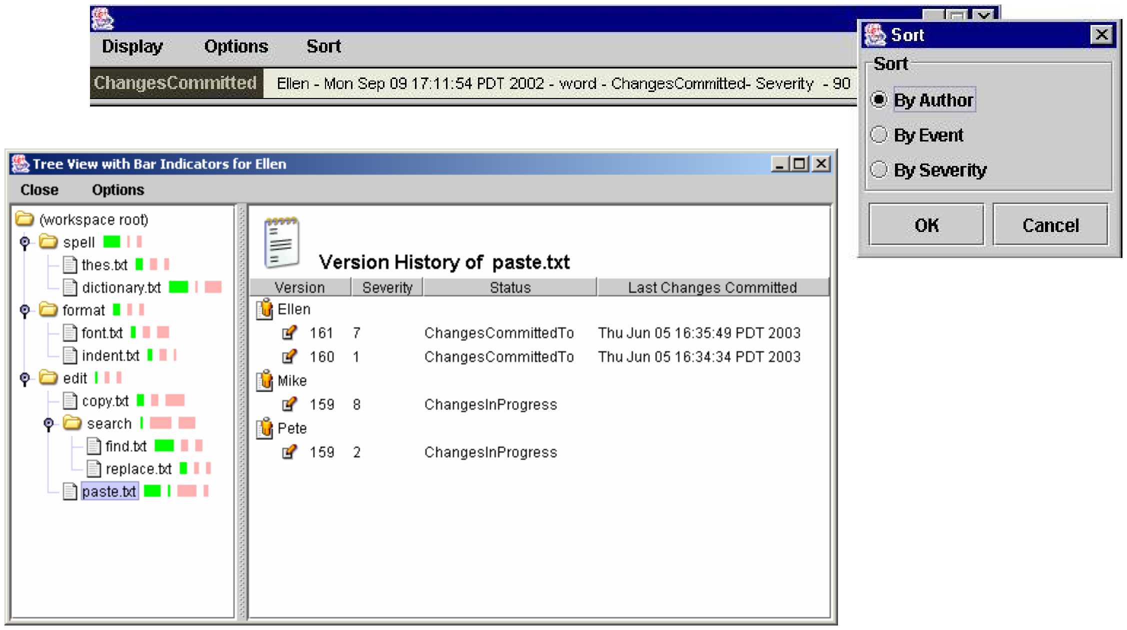
One of the core functions of any configuration management (CM) system is to coordinate access to a common set of artifacts by multiple developers who are all working on the same project. Existing CM systems address this task in two different ways: pessimistically and optimistically. In the pessimistic approach, a developer must lock artifacts before making any modifications. Such a lock prevents other developers from making concurrent modifications. In the optimistic approach, multiple developers can change the same artifacts at the same time. Conflicts may arise, but semi-automated differencing and merging tools help in identifying and resolving them.

Both the pessimistic and optimistic approach partition the work of developers in separate tasks to be performed in individual workspaces. These workspaces shield developers from the effects of other changes in other workspaces, but also have the unfortunate side effect of creating a barrier that prevents developers from knowing which other developers change which other artifacts in parallel. This regularly leads to problems, when conflicting changes are made on the same artifacts (direct conflicts), or when changing on one artifact by one developer do not “jell” with the changes by another developer on another artifact (indirect conflicts). In effect, the formal underpinnings of CM systems necessitate developers to look for other, additional mechanisms to coordinate their activities, as exemplified by the study in Section 2.

We have been building Palantír, a novel CM workspace awareness tool that embraces the continuous coordination paradigm. Palantír builds upon existing CM systems by hooking into their workspaces and sharing information about ongoing changes. It does so by intercepting workspace events, calculating a simple measure of severity to provide an assessment of the size of each change, distributing the event to the other workspaces in which the artifact is present, and graphically presenting the information to the developers “owning” those workspaces (Figure 3 shows two example visualizations). This information allows developers to actively self-coordinate: should they notice a potential conflict arising, or should they notice a developer is changing some file they intended to change or need in an unmodified form, they can proactively contact that developer, coordinate their actions, and avoid any larger problems down the road.

While a complete description of Palantír is beyond the scope of this paper (see [25]), we make two observations with respect to continuous coordination. First, we note that CM systems are inherently formal in nature, but that it was possible to extend them to also have an informal component. This aligns with our argument that continuous coordination does not require some radical approach, but rather relies in subtle but critical adjustments in work habits. In the case of Palantír, these adjustments are based on information that it shares with a developer regarding relevant ongoing parallel changes.

The second observation pertains to the fact that sharing information is only half the story of continuous coordination. CM tools can now be retooled to better support users in the kinds of actions they may want to take based



ChangesCommitted Ellen - Mon Sep 09 17:11:54 PDT 2002 - word - ChangesCommitted- Severity - 90

Sort: [● By Author] ○ By Event ○ By Severity

Version History of paste.txt

Version	Severity	Status	Last Changes Committed
Ellen			
161	7	ChangesCommittedTo	Thu Jun 05 16:35:49 PDT 2003
160	1	ChangesCommittedTo	Thu Jun 05 16:34:34 PDT 2003
Mike			
159	8	ChangesInProgress	
Pete			
159	2	ChangesInProgress	

Figure 3. Two example visualizations used by Palantír. One is a scrolling marquee, the other uses annotations in a file viewer to indicate local changes (green) and remote changes (red).

on the information. For instance, in Section 2 we identified the need for partial check-ins that exhibit only limited visibility. As another example, it should be possible to move some modified files to another developer's workspace so that it is possible for that developer to continue the work as part of their existing task (in case that task is deemed closely related to the one for which the initial changes were made). These examples require new kinds of functionality in CM systems, functionality aimed at a more fluid and flexible way of coordinating and collaborating in making changes.

7. Conclusions

Continuous coordination is a new collaboration paradigm that we believe will play an increasingly important role as we continue to develop new collaborative software engineering tools. It has the potential to overcome serious drawbacks associated with just taking a formal or informal approach, and provides users with both the tools and the information to self-coordinate their activities – all the while still guided by the overall process and supported by the facilities of the tools.

Our initial forays into building support for continuous coordination are promising. We have constructed a prototype versatile event service, YANCEES, that supports the different kinds of awareness needs that different software engineering tools have. We have an example of one such tool, Palantír, that combines a formal configuration management process with an informal mechanism for sharing information about ongoing workspace activities. Actual experience with the tools is limited at this point, but full-scale experiments are in the planning stages.

While the focus of our current work continues to focus on further enhancing YANCEES and Palantír with additional features, we also want to broaden our domain and start experimenting with continuous coordination in support of software design. Design is an inherently collaborative activity for which most tools support just a formal process; continuous coordination has the potential to significantly improve how designers coordinate their respective activities and interact with each other as they each contribute to the overall design.

Overall, a larger research agenda by the community is necessary to truly address and evaluate continuous coordination. We believe three canonical research questions must be answered:

1. When and how is it possible and desirable to combine a formal, process-based approach with an informal, awareness-based approach in support of continuous coordination? In effect, we need to understand the domains in which continuous coordination is an attractive solution.
2. What kind of generic infrastructure can be provided that new software engineering tools and environments can leverage? We need to examine not just generic event notification infrastructures, but also abstract from individual solutions to see if it is possible to reuse mechanisms via which information can be collected, organized, and presented.

3. What are the theoretical limitations of continuous coordination? Through extensive field studies, we need to determine what barriers to coordination and collaboration can be overcome with continuous coordination, and what barriers remain.

Acknowledgments

Effort funded by the National Science Foundation under grant numbers 0093489, 0205724, 0083099, and 0326105. Funding also provided by the Intel Corporation and through the Brazilian government under CAPES (grant BEX 1312/99-5).

References

[1] de Souza, C.R.B., Redmiles, D., Mark, G., Penix, J., Sierhuis, M. Management of Interdepend-encies in Collaborative Software Development, ACM-IEEE International Symposium on Em-pirical Software Engineering (ISESE 2003), September 2003a (to appear).

[2] de Souza, C.R.B., Redmiles, D., Dourish, P., Analyzing Transitions between Private and Public Work in Collaborative Software Development, International Conference on Supporting Group Work (Group 2003—Sanibel Island, FL), November 2003b (to appear).

[3] Button, G. and W. Sharrock, Project Work: The Organisation of Collaborative Design and Development in Software Engineering. Computer Supported Cooperative Work (CSCW), 1996. 5(4): p. 369-386.

[4] Grinter, R.E., Workflow Systems: Occasions for Success and Failure. Computer Supported Cooperative Work, 2002. 9(2): p. 189-214.

[5] Herbsleb, J.D., et al. An Empirical Study of Global Software Development: Distance and Speed. Proceedings of the International Conference on Software Engineering, 2001: p. 81-90.

[6] Barthelmess, P. and K.M. Anderson, A View of Software Development Environments Based on Activity Theory, Computer-supported Cooperative Work, Special Issue on Activity Theory and the Practice of Design, Vol. 11, No. 1-2, 2002, pp. 13-37.

[7] Nutt, G. The Evolution Towards Flexible Workflow Systems. Distributed Systems Engineering 3(4):276-294, December 1996.

[8] Perry, D.E., H.P. Siy, and L.G. Votta, Parallel Changes in Large-Scale Software Development: An Observational Case Study. ACM Transactions on Software Engineering and Methodology, 2001. 10(3): p. 308-337.

[9] Suchman, L., & Wynn, E. (1984). Procedures and problems in the office. Office: Technology and People, 2, 2, 133-154.

[10] Suchman, L. Plans and Situated Actions. CUP, Cambridge, 1987.

[11] C. Heath and P. Luff, Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Control Rooms. Computer Supported Cooperative Work, 1992. 1(1-2): p. 69-94.

[12] Dourish P. and V. Bellotti. Awareness and Coordination in Shared Workspaces. Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW '92), 1992: p. 107-114.

[13] W.C. Hill, et al. Edit wear and read wear. Proceedings of the Conference on Human Factors and Computing Systems, 1992: p. 3-9.

[14] Rosenblum, D. S., and Wolf, A. L. A Design Framework for Internet-Scale Event Observation and Notification. Proceedings of the Sixth European Software Engineering Conf./ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering, pp. 344-360, Sept. 1997.

[15] OMG, "Notification Service Specification v1.0.1," Object Management Group, 2002.

[16] R. E. Gruber, B. Krishnamurthy, and E. Panagos, "The Architecture of the READY Event Notification Service," in ICDCS Workshop on Electronic Commerce and Web-Based Applications, Austin, TX, USA, 1999.

[17] L. Lövstrand, "Being Selectively Aware with the Khronika System," presented at European Conference on Computer Supported Cooperative Work, Amsterdam, The Netherlands, 1991.

[18] M. Kantor and D. Redmiles, "Creating an Infrastructure for Ubiquitous Awareness," presented at Eighth IFIP TC 13 Conference on Human-Computer Interaction. Tokyo, Japan, 2001.

[19] B. Krishnamurthy and D. S. Rosenblum, "Yeast: A General Purpose Event-Action System," IEEE Transac-tions on Software Engineering, vol. 21, pp. 845-857, 1995.

[20] M. Mansouri-Samani and M. Sloman, "GEM: A Generalised Event Monitoring Language for Distributed Systems," presented at IFIP/IEEE International Conference on Distributed Platforms, Toronto, Canada, 1997.

[21] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," ACM Transactions on Computer Systems, vol. 19, pp. 332-383, 2001.

[22] G. Fitzpatrick, T. Mansfield, D. Arnold, T. Phelps, B. Segall, and S. Kaplan, "Instrumenting and Augmenting the Workaday World with a Generic Notification Service called Elvin," presented at European Conference on Computer Supported Cooperative Work (ECSCW '99), Copenhagen, Denmark, 1999.

[23] C. R. B. de Souza, S. D. Basaveswara, and D. F. Redmiles, "Using Event Notification Servers to Support Application Awareness," presented at International Conference on Software Engineering and Applications, Cambridge, MA, 2002.

[24] W3C, " XML Schema Part 0: Primer. W3C Recommendation," 2001.

[25] A. Sarma, Z. Noroozi, and A. van der Hoek, "Palantír: Raising Awareness among Configuration Management Workspaces", Proceedings of the Twenty-fifth International Conference on Software Engineering, May 2003, pages 444–453.