Computer Supported Cooperative Work **11:** 13–37, 2002.
© 2002 *Kluwer Academic Publishers. Printed in the Netherlands.*

A View of Software Development Environments Based on Activity Theory

P. BARTHELMESS & K.M. ANDERSON
Department of Computer Science, University of Colorado at Boulder, 430 UCB, Boulder, CO 80309-0430, U.S.A.

Abstract. We view software development as a *collaborative activity* that is typically supported by a software development environment. Since these environments can significantly influence the collaborative nature of a software development project, it is important to analyze and evaluate their capabilities with respect to collaboration. In this paper, we present an analysis and evaluation of the collaborative capabilities of software development environments using an activity theory perspective.

The discipline of software engineering (SE) emerged to study and develop artifacts to mediate the collective development of large software systems. While many advances have been made in the past three decades of SE's existence, the historical origins of the discipline are present in that techniques and tools to support the collaborative aspects of large-scale software development are still lacking. One factor is a common "production-oriented" philosophy that emphasizes the mechanistic and individualistic aspects of software development over the collaborative aspects thereby ignoring the rich set of human-human interactions that are possible over the course of a software development project.

We believe that the issues and ideas surrounding activity theory may be useful in improving support for collaboration in software engineering techniques and tools. As such, we make use of the activity theory to analyze and evaluate process-centered software development environments (PCSDEs).

Key words: activity theory, software development environments, software engineering

1. Introduction

Software development is essentially a *design activity*. Software is by nature intangible and its development occurs via a design effort – a stage that precedes construction in most other engineering disciplines. There is only one type of material, symbolic representations (Keil-Slawik, 1992). Design artifacts are created and transformed incrementally until a detailed description of a software product is attained.

Software Engineering deals with the building of software systems which are so large or so complex that they are built by a team or teams of engineers. Programming is primarily a personal activity, while software engineering is essentially a team activity (Ghezzi et al., 1991). Developing non-trivial software systems is therefore a task that requires a group of participants to work in concert. That is, they must *collaborate* in order to reach a common objective.

Early software development was based on the effort of individuals. The limited capacity of the existing machines prevented the development of truly complex systems and, as such, individual programmers were able to meet the needs of their users (typically themselves). As machines became cheaper and more powerful, the demand for complex software systems increased. Soon, individual programmers were not enough, and teams of programmers were needed to develop these new systems.

The transition from individual efforts to collaborative ones was (and is) not an easy one. Many projects failed because of a lack of understanding of large-scale software development. In particular, the artifacts used by individual programmers were not adequate to support the collaborative activities of large software teams.

As a result of these failures, practitioners turned their attention to the development of techniques for large-scale software development. New languages were developed, division of labor strategies were proposed, and new artifacts were suggested. A new term – software engineering – was coined to represent this effort, and the first software engineering conference was held in 1968 (Naur et al., 1976).

Process-centered software development environments (PCSDEs) form one category of software engineering tools. These environments are interesting in that they were designed to support the collaborative activities of large software teams. PCSDEs are an important collaborative tool, in the sense that they provide support for division of work, for anticipatory reflection via reified plans, and for operational support via the integration of applications. Therefore, they are potentially useful for mediating the collaborative aspects of a software development project.

Other authors have evaluated PCSDEs before, e.g. Ambriola et al. (1997). Our approach differs from these previous efforts in the sense that it is centered on a broader classification of collaborative activities, namely *activity theory* (Leontjev, 1978). The contribution of activity theory (AT) and similar theories is that they highlight perspectives that are different from the typical "production-oriented" view (Floyd, 1992) that is prevalent in software engineering.

Activity theory is a descriptive tool that is useful for analyzing and understanding collaborative work in general, independently of any specific field of application. We use AT in this paper to analyze PCSDEs with respect to their support for the demands of a collaborative effort. AT has been used as a source of inspiration in the context of CSCW by some authors, e.g. Bardram (1998a, b), Kuuti (1991, 1992), and Nardi (1996). Discussions of AT in the context of software development can be found in Floyd (1992).

The value of any theory is not "whether the theory or framework provides an objective representation of reality" (Bardram, 1998b), but rather how well a theory can shape an object of study, highlighting relevant issues. In other words, a classification scheme is only useful to the point that it provides relevant insights about the objects it is applied to.

Our aim is to explore the different alternatives explored by PCSDEs at a technical level, showing whenever possible the implications in the larger context of collaboration support. In order to ground our observations, we present a detailed survey of a representative set of PCSDEs and their associated features. Our detailed look at the features of PCSDEs illustrates both the diversity of options and the limitations of the employed approaches.

The rest of this paper is organized as follows. In Section 2, we briefly introduce the concepts of activity theory followed by a historical perspective on software development. The nomenclature of AT and PCSDEs are compared and contrasted in section 2.3. We then evaluate PCSDEs using an AT-based framework in Section 3. Finally, we present our conclusions in Section 4.

2. Background

2.1. A BRIEF OVERVIEW OF ACTIVITY THEORY

We now briefly introduce the concepts of activity theory that we employ in the rest of the paper. Readers interested in a broader treatment of the subject are referred e.g. to Leontjev (1978) and Engeström (1987).

The primary concept employed by AT is *human activity*. Activities, as defined by AT, provide enough contextual information to make an analysis meaningful, while avoiding a narrow focus on an individual or too broad a focus on whole social systems Kuuti and Arvonen (1992).

An activity is defined by an *object* – as in "object of the exercise" (Hasan, 1998). "An object can be a material thing, but it can also be less tangible (such as a plan) or totally intangible (such as a common idea) as long as it can be shared for manipulation and transformation by the participants of the activity" (Kuuti, 1996). An activity is motivated towards transforming an object into an *outcome*. Different objects define different activities. A *subject* is an agent that undertakes an activity. In collective activities, a *community* of agents share an object and work collectively on its transformation (Figure 1).

The abstract need to 'develop a system' motivates software development activities. A development team is a community that participates in this activity and shares its object – the evolving software system. Subjects, or agents, are the individual team members. The outcome of this activity is the developed system.

Central to AT is the concept of *mediation*. The relationships between *subject*, *object* and *community* are mediated by *tools*, *rules* and *division of labor*. These *artifacts* are used by a community to achieve a desired set of transformations on an object. Artifacts can range from physical tools, like a hammer, to psychological ones, like languages, procedures, methods and general experience (Bardram, 1998b).

In software development, a wide range of mediating artifacts is employed. These artifacts range from conventional languages (e.g., English) to specialized design and programming languages, methodologies, procedures and specific applications, such as editors, compilers, and environments. Membership in a development team is regulated by implicit or explicit rules. The division of labor emerges

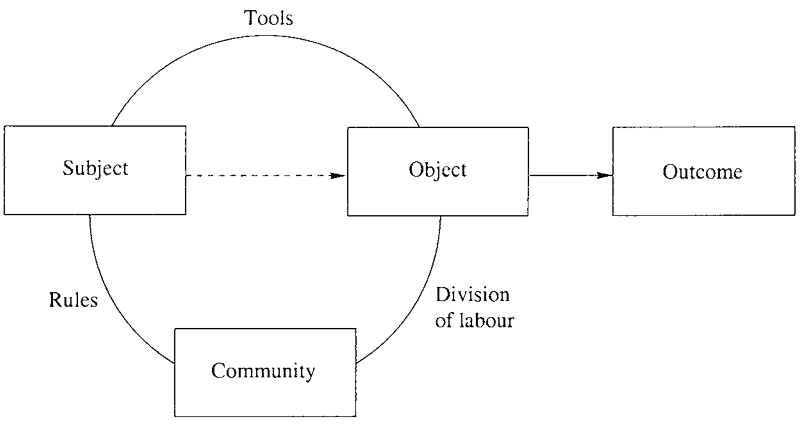

Figure 1. Entities and relationships in AT – based on Hasan (1998).

from the structure of the participating organizations, and can be further refined for a specific activity (e.g. "Mary is the leader of this project").

Mediating artifacts are the result of a historical-cultural process. Artifacts embody (crystallize) the collective experience of a community. Artifacts are therefore not static entities, but are continuously shaped and transformed to meet the evolving needs of the community (Bardram, 1997).

Activities transform objects via a process that typically has several steps or phases (Kuuti, 1996). Chains of *actions* guided by a subject's conscious *goals* carry out an activity over the short term, resulting in objective *results*. Actions themselves are realized through series of *operations*. Operations are performed automatically, in non-conscious fashion. There is not a strict one-to-one relationship between activities, actions, and operations – one activity is realized through many actions and one single action may be part of multiple activities. The situation is analogous for operations – an action may be carried through one or more operations, and a single operation may be part of multiple actions.

Software development starts with an extremely abstract object (e.g., "develop a new system") and is realized via a highly creative application of actions that build and transform knowledge representations that are shared by a team. Operations may involve, for instance, drawing a diagram or writing a specification.

With respect to our analysis, another useful tool is the notion of "subject-object-subject" relation that combines the object-subject and the subject-subject aspects of an activity. The former is referred to as the *instrumental* aspects of an activity, while the latter is known as the *communicative* aspects of an activity. The "subject-object-subject" notion was developed by Raeithel and Fichtner and adopted by Engeström in his analysis of the work performed in courts of law (1997). We base our description and nomenclature on Bardram's work (1998b).

Three different levels of interactions are suggested: *coordinated*, *cooperative* and *co-constructive* (called reflective communication by Engelström (1997).

The *coordinated* level corresponds to the routine, repetitive work performed by a group or organization. Coordinated work follows a pre-planned sequence with actors simply following their scripted roles. These scripts are "coded in written rules and places or tacitly assumed traditions. It coordinates the participant's actions as if from behind their backs, without being questioned or discussed" (Engeström, 1997). The underlying coordination ensures that the net result of these independent actions is the achievement of a common object.

The *cooperative* level involves the interaction of a group of agents. At this level, work is no longer independent. The actions of each agent influence the actions of others, enabling a *synergistic effect*. Actors "focus on the shared problem, trying to find mutually acceptable ways to conceptualize and solve it" (Engeström, 1997).

Co-construction (reflective communication) corresponds to the re-elaboration or re-construction of work practices. At this level, work itself is the subject of contemplation. New, better ways of doing it are devised. Co-construction can result in the redefinition of the organization and interaction in relation to a shared object.

There is a close interplay between these different levels, since they all represent aspects of what ultimately is a *collaborative activity*. A pattern of dynamic transformations between these levels can be observed: coordinated interactions can become cooperative and vice-versa; the result of a co-coordinated activity is the redefinition of the interaction itself, with a possible change in level. Take, for instance, a coding activity. Coding is typically conducted at the coordinated level, by an isolated programmer. If a programmer detects a problem with a specification of a task to be performed, he or she may consult with his or her peers and with his project leaders. The group can then decide to favor a specific interpretation, following a potentially complex sequence of collaborative interactions. The problem with the specification may also be related to the use of a particular technique or tool. In this situation, some co-construction may take place, potentially modifying the way the task itself is to be performed.

Therefore, it must be understood, that a level describes an *instantaneous* status of some activity, i.e., a routine interaction, usually performed at the coordinated level can become cooperative and then eventually co-constructive. Conversely, a typically cooperative level interaction can, through some co-construction effort, be reconceptualized and performed from that moment on at the coordinated level. One important aspect of activity theory is thus the recognition of the *dynamic transformation* between interaction levels.

To summarize, activities are performed by *subjects*, motivated by a goal, transforming an *object* into an outcome. An object may be shared by a *community* of actors, that work together to reach a desired *outcome*. *Tools*, *rules* and *division of labor* mediate the relationship between subjects, community and object. Activities are carried out by *actions*, which in turn are realized as sets of operations.

Coordination, *cooperation* and *co-construction* are three levels of subject-object-subject relations. Coordination focuses on individual actors, while cooperative activities focus on a shared object. Co-construction corresponds to re-elaborations of work practices. These three levels represent an instantaneous status of an activity.

2.2. A HISTORICAL PERSPECTIVE OF SOFTWARE DEVELOPMENT

Software development is characterized by its complexity. Few other human endeavors deal with entities that are as complex as large software systems. The object is typically very abstract. The outcome corresponds to the developed system and is reached via intricate symbolic manipulations of the original abstract object. Software is produced by a series of incremental transformations. The objective is to adhere to the initial motive in incrementally more detailed incarnations of a software system. Each step in a process should expand the concepts of previous ones, adding detail but preserving the originally intended semantics – this is known as preserving a system's *conceptual integrity* (Brooks, 1995).

Software development is a recent phenomenon. At first, very expensive machines were used mainly for military and scientific computations. The development of software was performed by scientists, using formalisms closely tied to a specific machine. It was common for a user of a system and a programmer of a system to be the same person. The complexity of the software was constrained by the (lack of) resources of the machines – programs simply could not be very large.

The rapid evolution of computer hardware made it viable to use computers outside of their initial domains. Large corporations took advantage of the increased processing power to aid in the process of conducting business and therefore started to develop software on their own. This transition was followed by the creation of a new generation of tools that made development easier. Compilers, editors, and operating systems were developed. Programming techniques (e.g. structured programming) started to appear.

As the scope and scale of software systems increased, the problems of large-scale software development appeared. When a software development task is complex enough to require the use of a large team, adhering to the original goal of a system becomes difficult. Team members are likely to have different backgrounds and perspectives. These perspectives must be aligned during the project, to avoid deviations from the desired goal. It is essential that all members work in concert to achieve the goal throughout the software development life cycle. This unity can only be achieved by a considerable amount of communication between the team members (Bardram, 1998b; Curtis et al., 1988).

The shift towards support for groups is not unique to software engineering. Grudin (1994) identifies this phenomenon as part of a broader groupware effort to provide support for large and small groups (corresponding to rings 2 and 3 of Grudin's classification).

Paradoxically, the communication that is vital for maintaining the conceptual integrity of a software system introduces some of the key problems encountered in such an effort, e.g., the "Tar-Pit" effect (Brooks, 1995). Communication overload can easily result from the indiscriminate (and late) addition of developers to a software development project. The root of the problem is that each participant of a project adds up to $n - 1$ communication channels (where n is the number of participants), leading to a combinatorial explosion of communication paths. This situation has, of course, a negative effect on progress, since soon all resources are spent on communication and none on the actual work, and of course, "work cannot be achieved by just talking about it" (Bardram, 1998b).

Initial large-scale software development efforts were chaotic, often resulting in an explosion of costs and development times, that were much larger than originally predicted. The outcome was typically quite different from the original objective of the activity. Many times the object was simply not produced and the effort was abandoned after the investment of considerable resources.

As a result, practitioners turned their attention to the development process itself. A new term, *software engineering*, was coined to represent this endeavor. Software engineering strives to further define techniques, processes, methodologies, and languages to ease the development of large software systems.

We can interpret the evolution of software development in AT terms as a quest for the appropriate mediating artifacts. *Tools*, *rules* and strategies for *dividing work* are essential concerns of SE. In this light, failures can be seen as activities that lacked the appropriate artifacts to mediate the relationships between subject, object, and community (see for instance the importance of representational conventions reported in Curtis et al. (1988)).

It may seem surprising that communication can be a root cause of significant problems. One must remember that the mediating artifacts for a particular domain are often the result of the work of many generations of practitioners of that domain. The rapid development of software engineering prevented it from having the opportunity of leisurely developing the appropriate set of mediating artifacts. This situation can be compared to a bridge design effort, in which thousands of designers try to reach an outcome without having a common understanding of the physics of bridge construction, without adequate shared languages, and without a clear division of work. Failures in this situation are hardly surprising.

Software engineering approaches the problem of collective development from a production-oriented viewpoint. While this view allows a certain amount of anticipatory reflection, e.g., the ability for subjects to predict important aspects of development prior to initiation and to assess results after completion (Floyd, 1992), it blinds software engineers to the communicative aspects of a collaborative software development activity.

PCSDEs represent yet another step in the evolution of mediating artifacts for software engineering. These environments are different from previous software engineering tools in that they are built explicitly to support some form of collaboration. To a limited extent, they incorporate a host of mediating artifacts. They incorporate reified procedures and rules, and enable the specification of the division of work. In addition, integrated tools provide access to an external shared memory. As a result, PCSDEs have the potential to be an extremely powerful collaborative tool, however the current generation of these environments must still overcome some significant shortcomings.

2.3. CONTRASTING ACTIVITY THEORY AND SOFTWARE DEVELOPMENT ENVIRONMENTS

PCSDE terms are often similar to those employed by AT, and yet can represent very different concepts. We now compare these terms in order to shed light onto their similarities and differences. We start by describing PCSDEs in their own terminology and then contrast these terms with those of AT.

A software production process is a sequence of steps or *activities* involved in building a software product. The order in which we perform these activities defines a life cycle (Ghezzi, 1991). A process model is an abstraction of the steps taken during software production.

PCSDEs are environments that provide support for the construction, evolution and enactment of *process models*. Process models can be considered software objects themselves and, as such, have *life cycles*, and are specified, designed, implemented, and deployed. Activities are the atomic units of work, and may be comprised of one or more steps. Activities are generally associated with *roles*. Roles describe in an abstract form the set of skills and/or responsibilities associated with the execution of one or more activities.

Process models are meant to be instantiated, resulting in an executable entity called a project. Zero or more projects based on the same or different process models can co-exist. Each instance or project can be in a different stage of its *life cycle*. Projects are enacted under the protection of an environment. This protection can vary widely, ranging from logging mechanisms, through guidance, to enforcement. An environment is typically responsible for keeping track of the progress of activities, their termination, and for launching new activities as soon as their preconditions are met. In addition, these environment can locate *agents* to fill specified roles.

Agents perform activities and can be either human participants or executable programs. During activity execution, agents create and transform *artifacts*.

Artifacts correspond to typical software development objects, such as requirements documents, test plans, test cases, etc.

Activity execution is facilitated through the environment. It activates any required tools of an activity in an integrated fashion. The environment is also responsible for supporting the management of the process, i.e., the *monitoring* and *adjustment* that is always necessary in the face of the variability and unpredictab-

Table I. Comparison of terms used by AT and PCSDEs

Term	Activity theory	PCSDEs
Activity	The unit of analysis	A single step in a process typically executed by a single agent
Object	The problem space motivating the activity	An entity (or an instance of an abstract data type in object oriented terms)
Artifact	Mediates the relationships between subject, object, and community	Document or other kind of deliverable
Tool	Physical or psychological device that mediates subject-object relationships	An application that is use to help perform some sequence of operations
Agent	An element of a community that shares an object	Human or application able to perform activities
Community	Subjects that share an object	Development team

ility of software development. Finally, the process itself is also subject to change, so meta-processes may be used to help in their evolution.

Table I summarizes the differences in nomenclature used by AT and PCSDEs.

As can be seen, the terms often have very different meanings in PCSDEs and in AT. In particular, an *activity* is taken to be an atomic element in PCSDEs, whereas in AT an activity is a composite entity with a broad scope. In PCSDEs, the closest match to AT's activity is a *process*. Activities in PCSDEs also lack the context they have in AT – they simply represent a predefined set of operations that needs to be carried out by an agent.

An *agent* is an entity defined by its capacity to perform some service ("execute an activity"), as opposed to humans that are part of a community with a shared object. There is no explicit notion of a community, even though one finds references to *development teams*.

The term *artifact* is typically used to refer to documents (or "deliverables") that are produced throughout a process. One can for instance have design artifacts, like a requirements document. *Tools* are used to perform operations on artifacts. These are, for instance, diagram and text editors, compilers, etc. Even though a diagram editor is considered a tool by PCSDEs, its diagramming language would not be, whereas in AT a language is considered a tool that is available to a community.

The term *object* has a rather fuzzy meaning, that we can take to be equivalent to "entity" in most contexts. It has also a technical meaning (an instance of an abstract data type).

References to these terms in the rest of the paper should be clear by context to refer to either the AT or the PCSDE meaning. In those cases where the context is ambiguous, an explicit reference will be provided. In addition, we will always have *activity* refer to the semantics defined by AT. Finally, even though *artifacts* and *tools* have a more nuanced meaning in AT, the terms are close enough to be used without further qualification.

3. PCSDEs as collaborative tools

PCSDEs are tools for storage and manipulation of process models that facilitate the creation of a software system and act as artifacts for the software development activity. Such artifacts can represent, to a limited degree, the procedures and rules that mediate a development effort, as well as some aspects of the division of work. This representation is typically limited to the most abstract and restricted sequence of independent work steps. PCSDEs typically do not venture into more complex forms of interaction. However, the results of these more complex interactions can be incorporated to a certain extent via the transformations of artifacts performed by agents as a result of decisions made outside the scope of a PCSDE proper, e.g., as a result of conversations held over lunch.

We analyze PCSDE features according to a framework based on AT. We characterize support features of PCSDEs according to their corresponding level of activity. We are interested in characterizing PCSDEs according to their ability to cope with the demands of collaboration. Aspects that are not directly related to collaboration are not considered, e.g. we do not consider which specific methodologies are supported.

We discuss, for each of the items of our assessment grid (see Table II), the general properties that must ideally be present in a PCSDE. The discussion is grounded by specific examples drawn from actual PCSDEs.

We are interested in both the instrumental and the communicative aspects of an activity and therefore choose as the main division of our framework the three level subject-object-subject notion of *coordination*, *cooperation* and *co-construction* (Bardram, 1998b; Engeström, 1997) described in section 2.1.

3.1. SUPPORT AT THE COORDINATED LEVEL

At the coordinated level, socially-sanctioned patterns of cooperation are exercised. Support at this level can be seen as providing "the molds or river beds in which each and every person unfolds his or her own version of each activity" (Raeithel, 1992). Recognizing and taking advantage of regularity and patterns in the work is an important part of an activity. "Anticipation is the motive of the activity, the goal of the action and orienting basis of the operation" (Bardram, 1997).

In the context of PCSDEs, coordinated level support is realized via the enactment of descriptions of chains of work steps, represented by a "process model" formalism. The representations of these patterns are used by the environment to constrain the work according to what is prescribed (aka enforcement). The effect

Table II. Analysis framework

Coordinated level support
Process Modeling Language
Organizational modeling
User Interaction
Interaction Paradigms
Enforcement
Tool integration
Cooperative level support
Communication
Shared artifacts
Concurrency control
Co-constructive level support
Transition between levels

of having a PCSDE interpret the patterns as scripts may cause undesired effects; for instance, the result of applying a sequence of steps in a process model may diverge from social convention. In situations where process enactment diverges from actual work practice, PCSDEs can become more of a hindrance to a community than a helpful tool. In addition, the malleability of the model may be exploited by a restricted group of stake-holders (e.g. managers), that may use it to try to impose their own personal agendas.

PCSDEs strongly emphasize support for the coordinated level. The support for the specification, definition and enactment of process models, i.e., of descriptions of process steps and their partial ordering, is arguably the main focus of PCSDEs. This is made explicit by the name chosen for these tools – *process centered* development environments.

The emphasis on the software process, and therefore on the coordinated level, is compatible with the production-oriented philosophy of software engineering. Focusing on the coordinated level allows environments to take maximum advantage of patterns and regularity in work practices. However, this emphasis on software process can result in "blindness" with respect to other important aspects of work, in particular collaboration. From a technical point of view, targeting the coordinated level makes complete sense – support for the other levels faces technical difficulties that are yet to be overcome, as we will examine in section 3.2.

The focus on the coordinated aspect through pre-specification of sequences of operations may cause as well *over-specification*. That is, the lack of adequate support for other modes of interaction may tempt a community to reduce all work into a sequence of operations, even when there is no clear pattern in the actual work practice. In other words, there might be a temptation to 'program' work in a similar way one builds other programs (which is very natural in a community of programmers).

Over-specification can cause frequent breakdowns due to a mismatch between a process model script and actual work practices. This is especially possible if the environment enforces strict sequencing, without allowing recombination and other changes that are necessary to adapt operations to specific situated conditions of work.

We next discuss specific coordinated level support features. We consider the elements of the *representation language* itself as well as the *organizational model* that binds agents to pre-specified *roles*. We also describe the *interaction a user has with an environment.*

3.1.1. *Process modeling languages*

The representation language, or process modeling language, is the main component in providing support at the coordinated level. It is a language that allows agents to specify and enact relevant aspects of routine work.

In general, the representation languages provided by PCSDEs provide extensive and flexible support for coordinated interactions. These languages share many characteristics with programming languages – some are in fact supersets of programming languages, e.g. APPL/A (Sutton et al., 1990). This situation is not surprising given the origin of PCSDEs, which are both developed and used by a community of software developers.

In each of these languages, there exists a trade-off between the level of abstraction and built-in functionality provided, and the flexibility a developer has in specifying a process. That is, either a language offers very flexible but low level support–that can be used to program a large set of desired behaviors – or higher level functionality is offered, at the expense of complete control over the final model.

One important characteristic of process modeling languages, identified by Ambriola et al. (1997), is how well does a language support the development of a process model. Models are only useful to the extent they are able to inform the understanding and refinement of current work practices. Furthermore, when a process model is enacted, its execution must support these practices, not hinder or disrupt them. As a result, there is a need for flexibility in both representing work in a manner suitable for discussion, i.e., in an abstract way, and for specifying the many details that are necessary for successful performance of the work itself.

Four basic conceptual elements are widely used across most systems as building blocks in process models (using PCSDE nomenclature): *activities* (work steps), *artifacts* (documents and other deliverables), *resources* (people, automatic agents, tools, schedules, and budgets) and *constraints* (temporal and organizational) (Huff, 1996; Lonchamp, 1994). All these aspects represent a rather high level of abstraction and must necessarily be complemented at the implementation level with additional elements, such as versions, communication paths, distribution, and workspaces. The choice of constructs shows how little of truly complex interactions can be represented by these languages. In particular, it reveals the operational, coordinated bias of the environments. What can be represented, and therefore enacted is just routine, repetitive work.

3.1.2. *Organizational modeling*

Organizational models are the element of process models that pertain to the division of work. The reader might be perhaps surprised by our choice of discussing this aspect under the general topic of coordinated level support. This choice is certainly not justified by the way AT deals with division of labor and its richer understanding of the interactions between a community and an object and between a subject and a community. Our choice is based on the very limited nature of the mechanisms available in PCSDEs, and is thus itself revealing.

Support for the division of work in PCSDEs is usually based on *roles*. Roles describe a set of prescribed scripted responsibilities (Biddle and Thomas, 1966). A participant can "play" many roles, according to his or her expertise. An *organizational model* maps participants to roles. Even though the importance of organizational models is recognized by some authors, e.g. Huff (1996) and Briand et al. (1995). Few details about actual PCSDE organizational mechanisms are available in the literature. This is surprising, given the complexity of most large organizations. Typically organizations employ elaborate hierarchies that are intermixed with other types of structures such as teams, committees, etc.

One significant concept that is lacking in current PCSDEs is the notion of *groups* of participants and their relationships. No kind of social awareness or stronger notion of community is present in these systems. As a consequence, their mechanisms are only usable at the coordinated level, where interactions between participants are not expected to occur. This support reduces mechanisms for distributing work to rigid pre-specified matching of participants to roles.

The lack of support for organizational modeling by PCSDEs has been recognized previously by Sommerville and Rodden (1996), who state that PCSDEs focus too much on mechanization and less on the fact that processes are extremely human-centered and support for humans and their organizational structures is critical.

3.1.3. *User interaction with the environment*

Participants interact with PCSDEs through a variety of mechanisms. In this sense, participants are *users* of an environment. Existing tools offer a relatively large set of alternative options that seem flexible in principle. There is also a clear notion about the implications of the levels of enforcement of strict sequences of operations. While the need for flexibility in the sequencing of chains of operations is understood, it is not clear that existing tools are prepared today to deal with such flexibility, with a few exceptions, e.g. Spade-1 (Bandinelli et al., 1994).

From the point of view of the participants, user interaction support is arguably the most important aspect. Participants are not typically concerned about models (unless they are engaged in co-construction activities). They are more concerned about how an environment might empower or constrain their work. We next discuss three aspects related to user interaction with an environment: *interaction paradigms* that may be employed, *enforcement policies*, and *tool integration*.

Interaction paradigms. Bandinelli et al. (1996) identify three interaction paradigms: *task-oriented*, *document-oriented* and *goal-oriented*.

– Task-oriented: this interaction style makes use of *agendas*. These agendas manage lists of relevant tasks for each user, as e.g. in SPADE-1 and Leu (Bandinelli et al., 1996). This paradigm usually implies strict sequencing of tasks.
– Document-oriented: the focus of interaction in these systems are documents and the services available over these documents. In Merlin, for instance, a *work context* displays in graphical form the relevant documents available for each role, as well as their inter-relationships. Available operations are provided via menus. Merlin "moves" documents between work contexts as tasks assigned to each role are completed (Junkerman et al., 1994). Users are usually free to apply enabled operations in any desired sequence.
– Goal-oriented: interaction with these systems is centered around a list of goals to be accomplished. In Marvel, for instance, these goals are the visualization of a set of rules a user can invoke (Bandinelli et al., 1996). Goals can usually be satisfied in any order, provided their preconditions have been satisfied.

While following a strict sequence of operations is natural for computers, users need to have control of the sequence of state transformations they are applying (Keil-Slawik, 1992). Operations have to be adapted to the concrete physical conditions of an action (Bardram, 1997). By letting users choose a sequence of operations, instead of having them prescribed by an environment, the intention of an action can be preserved in the presence of specific material conditions. A closely related concept is that of the *level of enforcement*, which we discuss in the next subsection.

Bandinelli et al. (1996) argue convincingly that no single paradigm can be considered a universal solution. There is an understanding that an adequate paradigm depends both on a user's preferences and the type of task at hand, and therefore should be flexibly determined. This is consistent with the complexity of activities. It would be surprising if a single paradigm would be sufficient for all situations.

Enforcement policy. An important aspect of process support concerns the amount of control exerted by an environment. On the one hand, an environment can be nonintrusive, keeping its presence hidden, as in SPADE-1 and Provence (Ambriola et al., 1997). Users activate tools directly and the environment keeps track of their progress in the background. On the other hand, systems can be totally prescriptive, enforcing a strict sequence of actions.

It has been recognized, though, that an overly prescriptive approach does not lead to good results. This is due to the amount of variability that is usually present in software development efforts (and work in general). Most systems therefore take a more careful stand, positioning themselves as guidance tools (Lonchamp, 1994). Furthermore, it appears to be the case that the level of enforcement also depends upon specific tasks. Even for a single process, it might be desirable to simply monitor some parts of the work, while strictly enforcing other aspects, perhaps because of security constraints, for example (Bandinelli et al., 1996).

We can again equate the levels of enforcement with AT's notion of implementation of actions through sets of operations that are adapted to situated contexts. The enforcement of specific sequences of operations obviously precludes this situated implementation of flexible sequences. Breakdowns then result from the inadequacy of tools. This is not a problem, and is part of the normal dynamics of work, except that the reconceptualization that should result from a breakdown sometimes cannot be reflected in the evolution of a tool, which would be a natural response. If the strict enforcement causing a breakdown is tightly built into a supporting application, changing the application is sometimes beyond what can be done by the participants themselves–they might for instance not have access to the source code of the application, or enough resources to implement the changes themselves.

Tool integration. Operations are typically executed through the activation of external tools. These external tools are automatically activated by an environment according to the specified steps of a process model. A typical mechanism for tool integration involves using a *wrapper* or *envelope* that serves as an interface between an environment and its tools, encapsulating the details of tool activation.

The amount of control that can be applied through this interface depends on the ability of each specific tool to react to external requests during execution. This is a problem especially with respect to legacy tools, that are not designed with integration in mind. In this case, it is only possible to control the activation of a tool and then wait for its termination. The problem with this approach is that an environment has no control over the actions of a tool during its execution. Thus, for instance, a text editor may load and modify many files without the knowledge of its environment.

Two approaches are used to more tightly integrate tools: either tools are specifically tailored to work with an environment, or some integration service is employed, e.g., Sun's Tooltalk (Sun Microsystems, 1997), DEC's Fuse (Digital Equipment Corporation, 1999) or OMG's CORBA (1999). In either case there are pros and cons. The integration of external tools enables the use of the latest versions, leveraging on the efforts of other developers. On the other hand, custom tools provide a tighter integration with an environment that enables better support for operations. In particular, since existing tools are typically developed for individual use, and not for groups, their use severely limits support for the cooperative level, which we discuss next.

3.2. SUPPORT FOR THE COOPERATIVE LEVEL

Cooperative interaction involves actors focusing on a shared problem, “trying to find mutually acceptable ways to conceptualize and solve it” (Engeström, 1997, p. 372). There is therefore a need for communication and sharing of an object. If we take design to be cooperative learning (Keil-Slawik, 1992) or cooperative problem solving (Fischer, 1994), support in PCSDEs for this level is essential. To support this kind of collaboration, participants need to be able to jointly apply exploratory manipulation of the symbols that are taken to represent a problem space (Curtis et al., 1988).

There are two obstacles to this kind of support. The first obstacle is the production-oriented philosophy of software engineering, discussed above. The basic premise that design *is* collaborative learning or problem solving is not compatible with this philosophy. The second obstacle is a technical one. The state of the art in supporting full-fledged collaboration is not yet sufficient to support large development efforts. This is not true only of PCSDEs: this goal is largely unsolved in CSCW in general. Problems range from efficient display of group work to individual agents (a user interface problem), to providing collaboration capabilities in a way that is compatible with existing social and work practices of a community of agents. The fact that external tools (editors, translators) integrated into an environment do not typically support collaboration aggravates the problem for PCSDEs.

Communication occurs at two levels – *explicit* and *implicit*. *Explicit* communication can be asynchronous (e.g. e-mail) or synchronous (e.g. video conferencing). *Implicit* communication is conveyed by changes in shared artifacts according to socially accepted conventions. Effective support for collaboration requires both implicit and explicit communication mechanisms (Robinson, 1993). In particular, there is a need for this support to be integrated: one should be able to modify a shared artifact, observe modifications performed by others, and also ‘talk’ explicitly about the intent of a modification as it occurs.

PCSDEs offer limited support for synchronous and asynchronous communication and provide some mechanisms for sharing artifacts. These capabilities are typically considered disjoint. That is, an agent either communicates or the agent modifies an artifact. It is typically not the case that implicit communication by artifact transformation can be accompanied by explicit simultaneous communication.

The access to shared artifacts provided by PCSDEs is predominantly asynchronous – participants can modify a shared artifact concurrently, but it is assumed they are working on their own, i.e., they are not coordinating their modifications. The term *conflict resolution* is employed to describe the situation when two or more participants commit changes on the same artifact simultaneously. As the term conveys, this situation is considered undesirable. However, these mechanisms used by PCSDEs go beyond the database concept of an *acid transaction*. Most employ mechanisms that are more compatible with the long duration that is typical in development transactions.

We next discuss communication and sharing support in more detail.

3.2.1. *Communication*

Asynchronous communication is enabled between agents via message exchange. Synchronous communication requires support for simultaneous interaction by all team members involved in a particular activity. Direct support for synchronous communication is missing from most systems.

A typical approach to supporting synchronous communication is to delegate responsibility to an external CSCW facility controlled by a PCSDE via tool integration (Ambriola et al., 1997). As a consequence, the cooperative interaction takes place outside the control of a PCSDE. It is, thus, no surprise that a misalignment may result between an environment's internal representation of a process and the reality of the external world with this approach.

The main problem with this issue is that PCSDEs are typically unaware of the communication that occurs between agents. Unfortunately, at the cooperative level, a fundamental part of an activity occurs via this communication. As such, PCSDEs have no means for incorporating or reflecting on this important part of a group's work.

Having agents collect and enter this missing information manually, as is sometimes suggested (Wolf and Rosenblum, 1993), is not adequate. Such a system is usually circumvented to avoid what is seen as extra work (Curtis et al., 1988). No agent can be expected to spend his or her time informing a system about work already performed (Suchman, 1983). One possible solution is to enhance an environment's functionality so that the work is performed within the environment, and not away from it. As mentioned by Wolf et al. (1993), this is easier said than done. There are, unfortunately, some very difficult problems to be faced in this area.

Few systems offer a tighter integration of synchronous communication:

– Spade-1 has been integrated with a CSCW toolkit (ImagineDesk) to include support for synchronous communication (Bandinelli et al., 1996). The architecture of the toolkit, that supports a service-request approach, matches the tool control facilities of Spade (black transitions), making a tighter integration possible. A process model fragment can be used to coordinate the preparation of the synchronous interaction (e.g., getting agreement on date and time, building an agenda). Detailed control of the interaction is attained though a process specification that sends service-requests to the toolkit components.

– Oz is possibly the only PSEE to include support for synchronous communication as a primitive construct of the modeling language (Bandinelli et al., 1996). Multiple participants can be specified and Oz makes each of them aware when a cooperation is about to start. The cooperation policy is fixed, i.e., it is not possible to specify alternative policies.

Finally, explicit communication is typically considered independent of operations on artifacts, which makes it difficult to support the double level of language that is required in such situations (Robinson, 1993).

3.2.2. *Sharing artifacts*

In design efforts, such as software development, it is essential to share work artifacts among team members. Artifacts direct attention to relevant details and offer a basis for "what if?" questions in complex scenarios (Robinson, 1993). The sharing of artifacts can also occur indirectly, as a result of coordinated level work being performed in parallel. Even though each activity might be executed in isolation, sharing will result if a common set of artifacts is transformed in two or more of these activities.

Unfortunately, the type of synchronous collaboration capabilities over artifacts that are available to a certain extent in CSCW tools are missing in PCSDEs. This lack of support precludes the use of techniques for collaborative design that may otherwise be useful (Fischer, 1994).

Our discussion therefore focuses on asynchronous collaboration support, that typically results from providing parallel access to shared artifacts during activities at the coordinated level.

– OIKOS (Montangero and Ambriola, 1994) offers a rich set of metaphors that handle different levels of cooperation: the *desk*, *environment* and *office*. A *desk* corresponds to a shared work space, potentially used by different roles to share information about the state of their work. Several agents may play their roles on the same desk, cooperating in a free fashion. Desks are part of *environments*. Several groups can play their roles in different desks, but under the same environment. The environment controls access to shared documents and therefore allows communication formalized by an environment's rules. Finally, an *office* groups many environments. The interaction of groups in different environments is the most formalized one and is controlled by a surrounding process.

– In ALF, several agents can share a single role, which allows them to work in the same working context, therefore sharing information. Different contexts can share artifact instances. Concurrency control is restricted to short term transactions, which is recognized by ALF's developers as a limitation (Canals et al., 1994, p. 161).

– EPOS offers a language (WUDL) to specify how access conflicts are resolved. According to the authors, "the goal is to keep cooperating and affected partners notified about product status and possible conflicts" (Ambriola et al., 1997, p. 300). This capability comes at a cost, however, since a desired behavior can only be achieved through careful programming.
– In Merlin (Junkerman et al., 1994), work is centered around the *working context* of each role. A working context presents a set of documents and their dependencies (as a graph) that are associated with a role, along with the activities that can be performed on each document. Cooperation is supported in an indirect way by Merlin, in that documents may "move" between the working contexts of different roles. In other words, when a document has been successfully modified by a role, it "disappears" from the role's context and is included in the working context of the role responsible for the next step of the process.

Access to the same artifact by two or more participants is controlled by concurrency control mechanisms, discussed next.

Concurrency control. This issue concerns the regulation of multiple agents accessing a shared set of artifacts. Software development stresses this issue in PCSDEs because it involves long-duration transactions and requires the sharing of large set of artifacts throughout the life cycle of a software system.

Concurrency control is traditionally (e.g. in the context of database systems) performed by locking and serializing access in such a way that *isolation* is guaranteed. Isolation guarantees that "each transaction is unaware of other transactions executing concurrently in the system" (Silberschatz et al., 1999). An approach that applies strict locking of shared artifacts for the entire duration of a transaction is unacceptable in PCSDEs largely because of the long-term nature of their transactions and the significant competition for access to artifacts by team members.

Some environments offer a rich set of mechanisms in the context of *extended transaction models* (Elmagarmid, 1992; Barghouti and Kaiser, 1991), e.g., EPOS, Spade-1 and Merlin. This functionality is typically restricted to pre-specified interactions that are programmed into a process model before process enactment begins. Strictly speaking, this reduces a cooperation-level activity to one that is completed as a coordinated-level activity. Transactional protection is based on mechanisms offered by an underlying database used by a PCSDE to store its artifacts.

3.3. SUPPORT FOR THE CO-CONSTRUCTIVE LEVEL

Co-construction corresponds to the "interactions in which the actors focus on reconceptualizing their own organization and interaction in relation to their shared objects" (Engeström, 1997, p. 373). The activity has a reflective object – some other activity – that is transformed to develop new possibilities of action (Raeithel, 1992).

Co-construction in PCSDEs is equated to reconceptualization of process models. Reconceptualization other than of the sequencing of operations has to be performed by means outside an environment proper. This is, in a sense, very limited. Breakdowns are generally seen as undesirable deviations from anticipated patterns, but are recognized as being unavoidable and intrinsic to the production of software (Ghezzi and Nuseibeh, 1998).

The construction of a process model is clearly at the co-constructive level. This kind of activity is motivated by the desire to define how some other process is to be conducted. Its object is this other process. In PCSDEs, activities of this type can be performed with the help of the environment. *Meta-processes* are used to empower and constrain the participants of the process development. Systems with such capability are said to be *reflective* (Ambriola et al., 1997).

Co-construction takes place during the execution of activities, as a result of breakdowns or because of deliberate reconceptualization of an object of work (Bardram, 1998b). Breakdowns and reconceptualizations are of course not restricted to the object of work itself, a software system being developed, but will also impact its associated development process. This situation occurs naturally, given the close relationship between process and product (Snowdon and Warboys, 1994). As a result, the usability of a system is largely determined by the way software and process evolution is supported.

Change seems to be intrinsic to the development of software, due to the fact that development is also (or primarily) a discovery process. At the start of a development project, little is known about the actual problem. As work progresses, an increased understanding may cause different approaches to become more desirable. Incremental delivery is one of the techniques that can be employed to make a problem quickly apparent. Internal problems need to be monitored and compared against expected progress. Once detected, a change needs to be implemented.

The absence of strong support for the cooperative level may be a hindrance during co-construction. Co-construction is inherently collective, and therefore it is questionable how helpful PCSDEs can really be if support for work is restricted to the coordinated level (Cugola, 1998).

Reflection is the method of choice for evolution in the majority of systems, e.g., EPOS, Spade-1, E3, Merlin, PWI, Peace (Lonchamp, 1994):

- In EPOS, processes can modify themselves at run-time through special operations of the modeling language, Spell (Ambriola et al., 1997).
- Spade-1 also supports dynamic evolution through reflective features, such as late binding and visibility of process information as data. Activity definitions and state of enacting instances of activities can be manipulated as any other kind of data (Ambriola et al., 1997).
- APPL/A (Sutton et al., 1990) and Process Weaver (Fernstrom, 1993) are exceptions, and only allow off-line evolution. Both systems are compiled, which makes impossible run-time modifications.

Changes have consequences, i.e., the impact of a change can affect to a greater or lesser extent other parts of a process. Certainly artifacts are impacted by changes, but even a process itself may suffer a transformation in response to a change.

Therefore, helping to determine the consequences of a change, or the propagation of a change, is an important task that must be supported by PCSDEs. Another aspect of dynamic change has to do with the fact that many *projects* may be enacting the same *process model*, and that their state may not be quiescent, i.e., one or more activities may be in execution at the time of the change (Huff, 1996).

3.4. TRANSITIONS BETWEEN ACTIVITY LEVELS

The concept of *dynamic transformation between levels* is central to the notion of hierarchical levels (Bardram, 1998b). In essence it establishes that there may be dynamic shifts in the mode of execution of an activity. An activity is not exclusively [performed at a] coordinated, cooperative or co-constructive [level]. It may present a predominant behavior, but a shift will probably occur, especially in a complex discovery process like software development. One implication of this is that the transitions between levels need necessarily to be fluid. In other words, there must be a natural progression from level to level. Ideally, this transition is imperceptible to agents.

The notion of hierarchical levels is alien to PCSDEs (as are most of the richer notions of AT). The production-oriented philosophy assumes that whatever has been coded in a process model corresponds to the steps that are necessary and sufficient during the unfolding of an activity. There seems to be an intrinsic belief in the stability of the means and object of work that would deem transitions unnecessary.

PCSDEs focus primarily on the coordinated level. It is expected that work will be conducted at this level at all times. Breakdowns or exploratory work that would be performed at the cooperative level are not supported in general. These have to be resolved outside the scope of an environment.

Depending on the nature of a breakdown, shifts to the co-constructive level can occur. This takes the form of a meta-process, a process whose object is another process. Given that co-construction probably demands a high degree of collaboration, it is questionable how successful these meta-models are in providing adequate support for work.

4. Summary and conclusions

Activity theory provides a rich descriptive tool that is useful for analyzing and understanding collaborative work in general, independently of any specific field of application. The basic unit of analysis in activity theory is a *human activity*. Activities, as defined by AT, provide enough contextual information to make an analysis meaningful, while avoiding a narrow focus on an individual or a too broad focus on whole social systems (Kuuti and Arvonen, 1992).

Very briefly, *activities* are undertaken by *subjects*, motivated by a purpose, transforming an *object* into an outcome. The object may be shared by a *community* of actors, that work together to reach a desired *outcome*. *Tools*, *rules* and *division of labor* mediate the relationship between subjects, community and object. Activities are carried out by *actions*, which in turn are realized as sets of operations. *Coordination*, *cooperation* and *co-construction* are the three levels of subject-object-subject relations. The focus of coordination is in each individual actor, while the cooperative level focuses on a shared object. Co-construction corresponds to re-elaborations of work practices. These three levels represent an instantaneous status of an activity.

Process-centered software development environments are tools built with cooperation in mind. They aim at supporting the work of (large) teams of developers. As the name implies, these environments are geared towards supporting a process.

To a limited extent, PCSDEs incorporate a host of mediating artifacts. They incorporate reified procedures and rules, and some of these rules specify the division of work. Integrated tools provide access to an external shared memory. PCSDEs approach the problem of collective development from a production-oriented philosophy (Floyd, 1992). This view is consistent with the heritage of the software engineering area. While this philosophy allows a certain amount of anticipatory reflection, it blinds software engineers to the communicative aspects of subject-subject relationships.

Support for the coordinated level is restricted to a shallow view of work as a sequence of operations, represented by a process model. Support for the division of work is closely tied to the distribution of tasks at the coordinated level.

Cooperative level support is restricted by the fact that implicit and explicit communication are treated separately. Sharing of artifacts is understood to be asynchronous, in the sense that it is assumed that actors are unaware of each other's existence, even if working on the same artifact concurrently.

Co-construction suffers from the process-oriented bias as well–only process models can be the object of reconceptualization. Since co-construction involves collaboration, the lack of stronger support at the cooperative level is limiting.

PCSDEs may be adequate for the downstream phases of development, but lack functionality for supporting upstream, design activities (Fischer, 1994). If one views design as collective problem solving or learning, then the lack of adequate cooperative level support is a serious hindrance.

The notion of the three levels of performance is not well represented in PCSDEs. As stated above, the production-oriented philosophy assumes that the routine steps in a process model are sufficient for the execution of an activity. As such, there is little support for transitioning an activity between the levels.

However, there is an implicit recognition that each activity may require different approaches; this recognition is apparent in the existence of tailorable process models. Evolution is also acknowledged as being intrinsic to an activity, and a set of specific (reflective) mechanisms is provided for that purpose.

The need for flexible and diverse user interaction paradigms is also a strong point of existing PCSDEs. It is understood that different users have different styles of work and that operations have to be adapted to unanticipated circumstances that require rearrangement and other kinds of modifications of the pre-planned sequence of operations.

Even though synchronous cooperative support is lacking, there is some effort to overcome the technical limitations that can be associated with the use of strict transactional mechanisms. Some cooperation is therefore possible through the extended transactional mechanisms provided.

PCSDEs suffer from the production-oriented philosophy of software engineering. Despite this, we have identified some positive aspects of these environments that provide support for work in less restricted ways. We believe that one contribution of this paper is to identify the utility of evaluating PCSDEs using the concepts of activity theory. We believe that future PCSDEs can benefit from this analysis and that support for the richer notions of collaboration that are provided by AT can lead to the development of more useful PCSDEs.

Acknowledgements

We gratefully acknowledge the contributions of our anonymous reviewers who helped to substantially improve our presentation of activity theory.

References

Ambriola, V., R. Conradi and A. Fuggetta (1997): Assessing Process-centered Software Engineering Environments. *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 283–328. http://www.acm.org/pubs/citations/journals/tosem/1997-6-3/p283-ambriola/.

Bandinelli, S., E. Di Nitto and A. Fuggetta (1996): Supporting Cooperation in the SPADE-1 Environment. *IEEE Transactions on Software Engineering*, vol. 22, no. 12. http://computer.org/tse/ts1996/e0841abs.htm.

Bandinelli, S., A. Fuggetta, C. Ghezzi and L. Lavazza (1994): SPADE: An Environment for Software Process Analysis, Design and Enactment. In A. Finkelstein, J. Kramer and B. Nuseibeh (eds.): *Software Process Modelling and Technology*. Taunton, Somerset: Research Studies Press Ldt., Chapt. 9, pp. 223–248.

Bardram, J. (1997): Plans as Situated Action: An Activity Theory Approach to Workflow Systems. In *European Conference on Computer-Supported Cooperative Work – ECSCW'97*. Lancaster, UK.

Bardram, J. (1998a): Collaboration, Coordination, and Computer Support: An Activity Theoretical Approach to the Design of Computer Supported Cooperative Work. Ph.D. thesis, Aarhus University. Daimi PB-533.

Bardram, J. (1998b): Designing for the Dynamics of Cooperative Work Activities. In *Conference on Computer-Supported Cooperative Work*, pp. 89–98.

Barghouti, N. and G. Kaiser (1991): Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, vol. 23, no. 3.

Biddle, B. and E. Thomas (eds.) (1966): *Role Theory: Concepts and Research*. New York: John Wiley & Sons, Inc.

Briand, L., W. Melo, C. Seaman and B. Basili (1995): Characterizing and Assessing a Large-Scale Software Maintenance Organization. In *Proceedings of the 17th International Conference on Software Engineering*, pp. 133–143.

Brooks, F. (1995): *The Mythical Man-Month: Essays on Software Engineering – Anniversary Edition*. Addison-Wesley.

Canals, G., N. Boudjlida, J. Derniame, C. Godart and J. Lonchamp (1994): ALF: A Framework for Building Process-Centered Software Engineering Environments. In A. Finkelstein, J. Kramer and B. Nuseibeh (eds.): *Software Process Modelling and Technology*. Taunton, Somerset, England: Research Studies Press Ldt., Chapt. 7, pp. 153–186.

Cugola, G. (1998): Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models. *IEEE Transactions on Software Engineering – Special Section on Managing Inconsistency in Software Development*, vol. 24, no. 11, pp. 906–907. http://church.computer.org/tse/ts1998/e0982abs.htm.

Curtis, B., H. Krasner and N. Iscoe (1988): A Field Study of the Software Design Process for Large Systems. *Communications of the ACM*, vol. 31, no. 11, pp. 1268–1286.

Digital Equipment Corporation (1999): DEC Fuse Home Page. http://www.digital.com/fuse/.

Elmagarmid, A. (ed.) (1992): *Transaction Models for Advanced Database Applications*. Morgan-kaufmann.

Engeström, Y. (1987) *Learning by Expanding: An Activity-theoretical Approach to Developmental Research*. Helsinki: Orienta-Konsultit Oy.

Engeström, Y. (1997) Coordination, Cooperation and Communication in the Courts. In *Mind, Culture, and Activity*. Cambridge University Press, Chapt. 28, pp. 369–388.

Fernstrom, C. (1993): Process Weaver: Adding Process Support to UNIX. In *Proceedings of the 2nd International Conference on the Software Process*, pp. 12–26.

Fischer, G. (1994): Domain-Oriented Design Environments. *Automated Software Engineering*, vol. 1, pp. 177–203.

Floyd, C. (1992): Software Development as Reality Construction. In C. Floyd, H. Züllighoven, R. Budde and R. Keil-Slawik (eds.): *Software Development and Reality Construction*. Springer-Verlag, Chapt. 3.2.

Ghezzi, C., M. Jazayeri and D. Mandrioli (1991): *Fundamentals of Software Engineering*. Prentice-Hall.

Ghezzi, C. and B. Nuseibeh (1998): Guest Editorial: Introduction to the Special Section. *IEEE Transactions on Software Engineering – Special Section on Managing Inconsistency in Software Development*, vol. 24, no. 11, pp. 906–907.

Grudin, J. (1994): CSCW: History and Focus. *IEEE Computer*, vol. 27, no. 5, pp. 19–27. http://www.ics.uci.edu/~grudin/Papers/IEEE94/IEEEComplastsub.html.

Hasan, H. (1998): Integrating IS and HCI Using Activity Theory as a Philosophical and Theoretical Basis. http://www.cba.uh.edu/~parks/fis/hasan.htm.

Huff, K. (1996) Software Process Modeling. In A. Fuggeta and A. Wolf (eds.): *Trends in Software: Software Process*. John Wiley and Sons, Chapt. 1.

Junkerman, G., B. Peuschel, W. Schäfer and S. Wolf (1994): MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment. In:A. Finkelstein, J. Kramer and B. Nuseibeh (eds.): *Software Process Modelling and Technology*. Taunton, Somerset, England: Research Studies Press Ldt., Chapt. , pp. 103–130.

Keil-Slawik, R. (1992): Artifacts in Software Design. In C. Floyd, H. Züllighoven, R. Budde and R. Keil-Slawik (eds.): *Software Development and Reality Construction*. Springer-Verlag, Chapt. 4.4.

Kuuti, K. (1991): The Concept of Activity as a Basic Unit of Analysis for CSCW Research. In: *Proceedings of the Second European Conference on CSCW*. Amsterdam, pp. 249–264.

Kuuti, K. (1996): Activity Theory as a Potential Framework for Human-Computer Interaction Research. In B. Nardi (ed.): *Context and Consciousness: Activity Theory and Human-Computer Interaction*. Cambridge, MA: MIT Press, Chapt. 2, pp. 17–44.

Kuuti, K. and T. Arvonen (1992): Identifying Potential CSCW Applications by Means of Activity Theory Concepts: A Case Example. In *Proc. of the Computer Supported Cooperative Work – CSCW'92*, pp. 233–240.

Leontjev, A. (1978): *Activity, Consciousness and Personality*. Englewood Cliffs: Prentice-Hall.

Lonchamp, J. (1994): An Assessment Exercise. In A. Finkelstein, J. Kramer and B. Nuseibeh (eds.): *Software Process Modelling and Technology*. Taunton, Somerset, England: Research Studies Press Ldt., Chapt. 13, pp. 335–356.

Montangero, C. and V. Ambriola (1994): OIKOS: Constructing Process-Centred SDEs. In A. Finkelstein, J. Kramer and B. Nuseibeh (eds.): *Software Process Modelling and Technology*. Taunton, Somerset, England: Research Studies Press Ldt., Chapt. 6, pp. 131–152.

Nardi, B. (ed.) (1996): *Context and Consciousness: Activity Theory and Human-Computer Interaction*. Cambridge, MA: MIT Press.

Naur, P., B. Randell and J. Buxton (eds.) (1976): *Software Engineering: Concepts and Techniques*. New York: Peterocelli/Charter.

OMG (1999): Corba Home Page. http://www.corba.org.

Raeithel, A. (1992): Activity Theory as a Foundation for Design. In C. Floyd, H. Züllighoven, R. Budde and R. Keil-Slawik (eds.): *Software Development and Reality Construction*. Springer-Verlag, Chapt. 8.4.

Robinson, M. (1993): Design for Unanticipated Use. ... In C.S.G. de Michelis and K. Schmidt (eds.): *Proceedings of the Third European Conference on Computer-Supported Cooperative Work*. Milan, Italy, pp. 187–202.

Silberschatz, A., H. Korth and S. Sudarshan (1999): *Database System Concepts*, 3rd edn. McGraw-Hill.

Snowdon, R. and B. Warboys (1994): An Introduction to Process-Centered Environments. In A. Finkelstein, J. Kramer and B. Nuseibeh (eds.): *Software Process Modelling and Technology*. Taunton, Somerset, England: Research Studies Press Ldt., Chapt. 1, pp. 1–7.

Sommerville, I. and T. Rodden (1996): Human, Social and Organisational Influences on the Software Process. In A.Fuggeta and A. Wolf (eds.): *Trends in Software: Software Process*. John Wiley and Sons, Chapt. 4.

Suchman, L. (1983): Office Procedure as Practical Action: Models of Work and System Design. *ACM Transactions on Office Information Systems*, vol. 1, no. 4, pp. 320–328.

Sun Microsystems (1997): Tooltalk User's Guide. ftp://192.18.99.138/-802-7318/-802-7318.pdf.

Sutton, S., D. Heimbigner and L. Osterweil: (1990): Language Constructs for Managing CHange in Process-centered Environments. In *Proceedings of the 4th Symposium on Practical Software Development Environments*.

Wolf, A. and D. Rosenblum (1993): Process-Centered Environments (Only) Support Environment-Centered Processes. n: *Proceedings of the 8th International Software Process Workshop (ISPW8)*. Wadern, Germany, pp. 148–149.