# Let’s Talk About Bugs!

**Peter H. Carstensen & Carsten Sørensen**
*Systems Analysis Department, Risø National Laboratory*
*DK-4000 Roskilde, Denmark, {phc, carsten}@risoe.dk*

**Tuomo Tuikka**
*Department of Information Processing Science, University of Oulu*
*SF-90570 Oulu, Finland, ttu@rieska.oulu.fi*

**Abstract**

*Software testing is often a complex process potentially involving a large number of geographically distributed people with different perspectives and competencies. Software testers, software developers and project managers engage in discussions about the software errors found, they negotiate the relative importance of the bugs, they allocate responsibilities and resources, they coordinate who is doing what, etc. They talk about bugs. In order to coordinate and manage talking about bugs, a number of means for coordination are applied. The aim of this paper is to analyze coordination work in software testing in order to promote general requirements for computer support. We have studied the testing of more than 200,000 lines of code at Foss Electric, a Danish manufacturing company, and focused on two aspects: Firstly, the coordination activities related to the process of distributed registration, classification, diagnosis, correction, and verification of software errors, as well as the monitoring of the state-of-affairs of testing activities. Secondly, the mechanisms used to support the coordination. The analysis resulted in the identification of the need for computer support for coordinating this part of the software testing process, e.g., support of distributed classification, routing of information, and facilities providing an overview of state of affairs.*

## 1. Introduction

"The lay public, familiar with only a few incidents of software failure, may regard them as exceptions caused by inept programmers. Those of us who are software professionals know better: the most competent programmers in the world cannot avoid such problems. […] Software is released for use, not when it is known to be correct, but when the rate of discovering new errors slow down to one that management considers acceptable. […] It is not unusual for software modifications to be made in the field. Programmers are transported by helicopter to Navy ships: debugging notes can be found on the walls of trucks carrying computers that were used in Vietnam. It is only through such modifications that software becomes reliable."

(Parnas 1985)

Software testing is an extremely complicated activity. In practice an exhaustive test is impossible (Myers 1979, Parnas 1985). Despite of all the techniques and methodologies for specific source code testing, black box testing, usability testing, etc., no methodologies exists for establishing a set of unambiguous criteria for a sufficient test strategy in order to ensure that the product is reliable, usable, and correct (Petchenik 1985). Hence, much effort is required to establish a common understanding among software developers, software testers, and software managers of when a product is acceptable. Organizations involved in software testing typically apply the strategy of having people with different skills and perspectives test the software. As argued by Dahlbom & Mathiassen (1993): "Effective quality control requires a certain division of labor and responsibilities. In practice, quality is not the only concern, and there is a constant struggle between quality and resource interests. Independence is needed to constantly defend a quality position and to avoid the self-deception in having systems developers evaluate their own products." The division of labor in the software testing process can either be done so that different actors perform different subtasks (detection, diagnosis, correction, etc.), or it can be organized so that each person has the responsibility for all testing activities within a limited part of the program. In any case, the participants will inevitably be mutually interdependent. As argued by Parnas (1985), the need for coordination in software development can not be eliminated by structuring the software properly. In order to mesh their work results, interdependent actors performing distributed software testing tasks must coordinate and negotiate their work (Schmidt 1994, Kraut & Streeter 1995).

Recent works have addressed cooperative aspects of software development. Johnson & Tjahjono (1993) promote the CSRS system supporting collaborative software review, and Mashayekhi *et al.* (1993) describe the CSI system supporting distributed collaborative software inspection. Swenson *et al.* (1994) show by example how a workflow system can support the coordination of distributed software testing. Kraut & Streeter (1995) present an analysis of formal and informal aspects of coordination in the software development process, focusing on the coordination conducted by peer-to-peer communication, by project schedules, and by review and inspection meetings. They argue for the importance of informal direct communication in systems development, but at the same time argue that the excessive transaction costs and the ephemeral nature of the information transferred in informal communication implies that more formal coordination means must be applied. We focus is on how formal means can support coordination of distributed software testing activities.

We have studied coordination work in the testing of more than 200,000 lines of code at the Danish manufacturing company Foss Electric. Our analysis focused on the aspects of coordination supported, stipulated and mediated by various paper- and computer-based coordination tools, e.g., a paper-based bug handling workflow system, a centralized binder containing bug forms, a software module integration procedure, and a project resource schedule. More specifically, we have analyzed how these tools supported the coordination of distributed registration, classification, diagnosis, correction and verification of software errors. By stipulating who is doing what, the tools provide persistent accounts to support the more ephemeral coordination work. This paper does no present an analysis of the of software testing in general.

Based on the analysis, we have identified needs for computer support for this part of the software testing process, e.g., support of distributed bug classification, routing of information, monitoring of state of affairs, etc. The purpose of applying such computer based coordination tools is by no means to remove the need for peer-to-peer communication and review meetings. It is rather to provide means of coordinating a multitude of detailed decisions which can form the basis for talking about bugs.

The next section describes the research approach applied. Section 3 gives an overview of the field of software testing. Section 4 presents the Foss Electric case. Section 5 discusses the need for computer support of the articulation of distributed registration, classification, diagnosis, correction, verification of software errors, and for monitoring the state of affairs of the testing process. Section 6 discusses our results.

## 2. Research Approach

This paper is based on data collected in an empirical study of one development effort at Foss Electric. The study focused both on the coordination of software testing and on the coordination of engineering design and process planning. This paper only reports on the coordination of software testing. The field study and the preliminary data analysis lasted a total of six months and was exclusively based on qualitative data collection techniques such as qualitative interviews (Patton 1980), observations, study of project documentation, and participation in project meetings. A total of 21 interviews were conducted, and we participated in 10 project meetings. Approximately 75 man-hours were spent observing the development process. The approach was inspired by perspectives promoted in several research efforts (cf. Bucciarelli 1984, Schmidt & Carstensen 1990). As argued by Siemieniuch (1992), field studies are important in order to obtain a coherent understanding of how computer tools can support product development in a manufacturing setting. The data analysis was based on theories and conceptualizations from the field of

Computer Supported Cooperative Work (CSCW), as promoted in Schmidt (1994).

Yin (1989) distinguishes between a case study approach and an ethnographic approach. The former being structured and targeted, and the latter being more unstructured and primarily based on observation. Our approach can be characterized as having elements from both types with a predominant case study bias. Although we did not start out with a strict set of hypotheses, we did bring an articulated perspective. The purpose of the study was to investigate how various paper-, board- and computer-based mechanisms supported the coordination of distributed work (Sørensen *et al.* 1994, Carstensen *et al.* 1995).

A qualitative approach offers the obvious strength of providing rich and detailed data, enabling a deep understanding of the conditions under which work is performed. It does, however, present a major limitation in terms of promoting statements of general validity. As Mason (1989) argues, the purpose of research must be to provide both the richness of detail and relevance of research problems studied, as well as a certain tightness of control or rigor. We do not believe that one empirical effort necessarily needs to encompass both aspects. We do, however, recognize that since the results reported in this paper are drawn from a single field study, we can neither make claims as to the generality of the findings, nor to a rigorous research approach. The organizational culture at Foss Electric favors both individual and group achievements, work is primarily organized in projects and it is not a particularly hierarchical organization. Various coordination systems were both designed and used by project members without leading to conflicts or fear of being monitored. It is, therefore, reasonable to assume that the types of phenomena studied at Foss Electric can only be made subject to generalization if an organizational culture of a similar nature is observed.

## 3. Software Testing and Its Coordination

"That was back on Mark I. It was in 1945. We were building Mark II—and Mark II stopped. We finally located the failing relay and, inside the relay, beaten to death by the relay contact, was a moth about three inches long. So the operator got a pair of tweezers and carefully fished the bug out of the relay and put it in the log book. He put scotch tape over it and wrote, "First actual bug found." And the bug is still in the log book under the scotch tape and it is in the museum of the Naval Surface Weapons Center at Dahlgren, Virginia."

Grace Murray Hopper quoted in Jennings (1990)

"The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it is a disguise that the error is the programmer's own creation."

(Dijkstra 1989)

The understanding of bugs in programs has changed since the 1940's, although the idea of having an error in a program is as unpleasant as having a bug inside the computer. The metaphor describing a software error as a bug is confusing. An error can cause reactions as if it was a living insect that should be removed by using poison while programming, but it is, of course, created by the programmer and should as such be regarded as a software error. The word bug is stuck in our language and it is probably as difficult to get rid of as errors in software. We will call them bugs since our focus is more in talking about them, than on studying formal techniques for finding them.

The art devoted to finding software errors is called software testing. Myers (1979), for example, defines software testing simply as being the process of executing a program with the intent of finding errors. The view to software testing has been changing during the last decades. The early view of programming and testing was that you "wrote" a program and then you "checked it out." Later testing has been defined as evaluation of software or prevention of problems. An illustration of the evolution in software testing can be found in Gelperin and Hetzel (1988). Hetzel (1988) for example defines software testing as any activity aimed at evaluating an attribute or capability of a system, and determining that it meets its required results. It is widely accepted that software testing is one of the corner stones of modern software engineering and thus should be an integrated part of any quality program, and a central activity in all quality assurance groups (Yourdon 1988).

The field of software testing spans mathematical theory, the art and practice of validation, and methodology of software development (Hamlet 1988). Software testing literature mostly addresses the relationship between the testing and the software engineering process (i.e., the use of testing methods and tools). The cooperative aspects of the process is only marginally addressed. It is important to consider software testing as a part of the software engineering process, i.e., relate the stages in the process to stages of testing, e.g., unit, integration, system, product, customer, and regression testing (Dalal *et al.* 1993). A broad array of testing methods and techniques are available today, e.g., black and white box testing techniques providing a systematic approach to the design of test cases (see for example Beizer 1990 and Hetzel 1988). A fast growing flux of automated software testing tool products influences the field today. Neither the stages, the techniques, or the automated tools will, however, be discussed further in this paper.

The vast majority of software organizations have substantial room to improve the manner in which testing is conducted and software testing is often the poorest scheduled part of programming (Brooks Jr. 1982, Pressman 1988). If the importance is not recognized correctly, the project planning will not include enough time. This causes many problems to the testers, i.e., accumulated pressure in the work. The complexity of the testing work and a tight schedule influence on the decisions determining whether or not a software product meets its requirements. That is, there is no systematic way to search, no way to judge points selected, and no way to decide when to stop (Hamlet 1988).

Since exhaustive testing is impossible (Myers 1979, Parnas 1985), a common understanding of the status of a software product can only be established by means of negotiations. These situations are not easy to cope with and will often result in work settings including many cooperating actors. The actors in an ensemble developing and testing software become interdependent: "Cooperative work occurs when multiple actors are required to do the work and therefore are mutually dependent in their work." (Schmidt 1991). In order to handle the underlying interdependencies among the cooperating and mutually interdependent actors, a set of "second order activities" is needed. The actors must, in often complex ways, coordinate the plurality of tasks and relationships between actors and tasks. We use the term coordination main as the direction of individuals efforts towards achieving commonly and explicitly recognized goals and "the integration or linking together of different parts of an organization to accomplish a collective set of tasks" (Kraut & Streeter 1995). Notice, that by this definition testers, designers, etc. do not necessarily share goals (Bannon 1993). In our terminology, coordination can include activities aimed at negotiating, establishing, maintaining, and refining the conceptual structures and salient dimensions along which the coordination must be conducted (Schmidt 1994).

## 4. The Foss Electric Case

> "With the number of developers involved, it is extremely important that all problems are registered, otherwise they just 'disappear' […] An important derived product then, is a list of problems reported fixed but not yet tested. Based on the lists and the problem descriptions, the platform master can check and then report the problem corrected."
>
> (Software designer at Foss Electric)

Foss Electric is the largest manufacturer of highly specialized equipment for analytically measuring quality parameters of agricultural products in the World. The company is localized in Denmark with service and distribution offices in many countries. They employ approximately 700 people. The customers are laboratories, slaughterhouses, dairies, etc. Foss Electric is a matrix organization, and development of new instruments is organized in projects which typically include specialists with design competence in the fields of mechanical-, electronic- and software design, as well as in optics and chemistry.

### *4.1. The S4000 Project*

The objective of the S4000 project was to build an instrument for analytical testing of raw milk. It included functionality which previously had been placed in several instruments, and furthermore introduced measurement of new quality parameters of milk. The instrument consists of approximately 8000 components grouped into a number of functional units, such as: Cabinet, pipette unit, conveyer, PC, other hardware, flow-system, and measurement unit. The S4000 was the first product featuring a built-in an Intel-based 486 PC. Configuration and operation of the instrument is done through a Windows user interface, i.e., the user-instrument interaction is based on a graphical user interface and use of mouse and keyboard. Version 1 of the software contained approximately 200,000 lines of source code. At most 50 people at a time were directly involved in developing the instrument, and the project lasted approximately 2 1/2 years. The core personnel involved in the design included a number of designers from each of the areas of mechanical design, electronic design, software design, and chemistry. In addition there was a handful of draught-persons and several persons from each of the departments of production, model shop, marketing, quality assurance, quality control, service, and top management. A group of between 5 and 12 software designers was involved in designing and coding the software required to operate and control the S4000.

### *4.2. Coordinating the Bug Handling Activities*

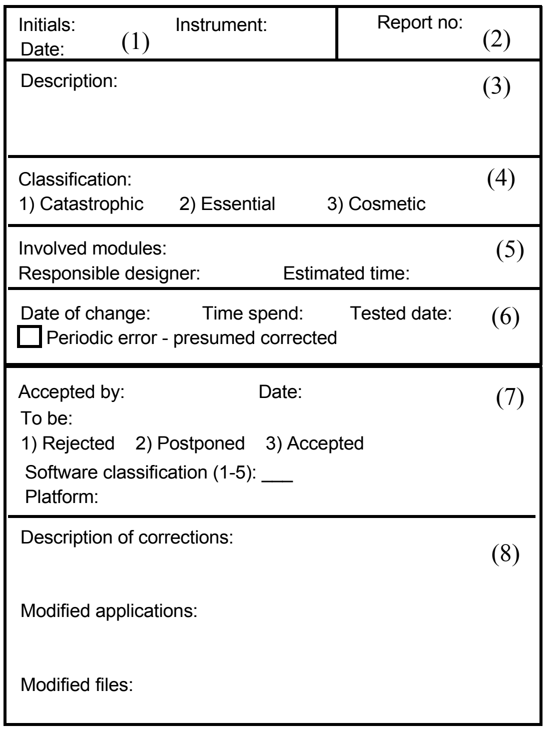
During the S4000 project, the software designers realized problems in coordinating, controlling and monitoring the software testing activities. This and external requirements for more precise measurements of the status of the software testing process led to a standardized bug form and a centralized binder being invented, used, and refined by the involved software designers during the S4000 project for registering and filing identified bugs. Furthermore, a set of concomitant procedures and conventions for the use of the form and binder were established. Some of these were formulated by the designers themselves and written down as organizational procedures, others were established as conventions agreed upon by the software designers. The bug form and the general procedure for using the form are illustrated in Figure 1. The purpose of the form and the procedures were to ensure that all identified bugs were registered and "remembered" until they were corrected. This was accomplished by ensuring that each registered bug was represented by one form only, and by ensuring that changes to the state of the process dealing with a bug was reflected in the form representing the bug. To ensure that all bugs were handled, a central file (the binder) contained a copy of the form until a final state was reached, and the original form was filed. Several groups of actors and roles were involved in this process, i.e., users of the bug form, the binder, and the procedures. These were:

- Testers from different departments and with different perspectives on software quality involved in testing the S4000 software. Apart from the software designers approximately 20 other actors were involved in testing the software.
- The spec-team, a group of three software designers responsible for diagnosing bugs and deciding how to handle the correction of bugs. These persons represented different areas of expertise in relation to the software architecture.
- Software designers each responsible for one or more software modules.
- The central file manager who was one of the software designers responsible for maintaining a binder containing forms for all registered bugs.
- The platform master responsible for managing and coordinating the activities in one of the integration periods (called a platform period).
- The plan-manager responsible for updating the work plans. In the S4000 project the plan-manager was one of the spec-team members.

The routing of the bug forms among the six roles were done according to the following eight steps (see Figure 2):

1. A tester sends a form to the spec-team describing a registered and classified bug

FIGURE 1. **The bug form used at Foss Electric and the general procedure followed when using the forms for registering, diagnosing, and correcting bugs. CFM means 'central file manager' and PM is 'platform master'. The form is a sheet of A4 paper printed on both sides. The numbers indicate who fills in which information in the form**


Initials: Instrument: Date: (1) | Report no: (2)

Description: (3)

Classification: (4)
1) Catastrophic 2) Essential 3) Cosmetic

Involved modules: (5)
Responsible designer: Estimated time:

Date of change: Time spend: Tested date: (6)
☐ Periodic error - presumed corrected

Accepted by: Date: (7)
To be:
1) Rejected 2) Postponed 3) Accepted
Software classification (1-5): ___
Platform:

Description of corrections: (8)

Modified applications:

Modified files:

The actors fill (or add information) in:

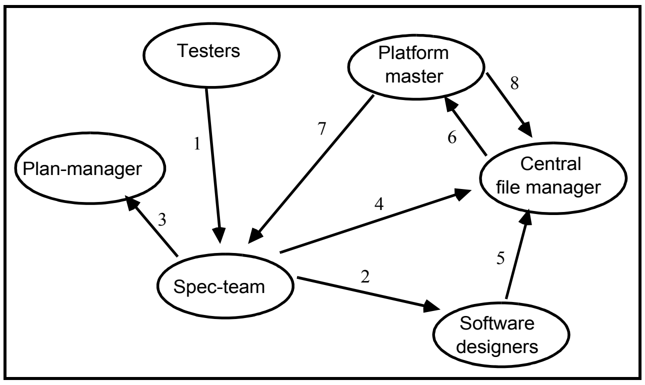
The testers: (1), (2), (3), and (4)
The Spec-team: (3), (4), (5), and (7)
The designers: (6) and (8)

The procedure for handling bugs:
- A tester register and classifies a bug (field 1,2,3, and 4)
- The tester sends the form to the spec-team
- The spec-team diagnose and classify the bug (field 3, 4 and 7)
- The spec-team identifies the responsible designer (field 5)
- The spec-team estimates the correction time (field 5)
- The spec-team incorporates the correction work in the work plans
- The spec-team requests the designer to correct the problem
- The designer corrects the bug and fills in additional correction information (field 6 and 8)
- The designer sends the form to the central file
- The CFM sends the form to the PM and insert copy in central file
- The PM verifies the correction
- The PM returns the form to the central file

2. The spec-team adds diagnosis and estimation information and sends it to a software designer
3. The spec-team requests the designer responsible for the plans to update the planning spread-sheet
4. A copy is sent to the central file manager. If a bug is rejected the original is sent to the central file manager
5. The software designers add correction information to the form and send it to the central file manager
6. The central file manager sends a pile of forms to be verified to the platform master
7. Forms which can not be verified are send to the spec-team
8. Verified forms are send to the central file manager

All registered bugs were filed in a binder providing all software designers and other testers with access to the state of affairs in the testing process. During one and a half year approximately 1400 bugs were registered, treated and filed in the S4000 project. The binder was physical-

**FIGURE 2. The roles and the stipulated flow of bug forms between them in S4000 software testing. Arrow numbers refer to the procedure presented in the text**



ly placed in the room use by the project team (all the designers, but not necessarily the testers). The binder had the following seven entries reflecting the status of a specific bug, and in each of these entries the forms were filed in chronological order:

1. Non-corrected catastrophic bugs (copies)
2. Non-corrected essential bugs (copies)
3. Non-corrected cosmetic bugs (copies)
4. Postponed bugs (originals)
5. Rejected bugs (originals)
6. Corrected bugs not yet verified (copies)
7. Corrected bugs (originals).

The bugs reported in the S4000 project were handled according to the twelve procedural steps listed in Figure 1. In most cases the prescribed procedures were followed strictly. There were, of course, situations in which the actors did not follow the procedure. A thorough step by step description of the procedure and of the "typical" exceptions is given in Carstensen (1994).

An interesting and important characteristic of the software development and testing work in the S4000 project was the organization of software development and the structuring of plans in working cycles called "platform periods". A platform period was typically 3–6 weeks of development followed by one week of integration. Version 1 of the S4000 system covered approximately 15 platform periods. As a configuration control measure, revisions of the software were only allowed after the "platform" had been released. For each platform period, a platform master was appointed by the group of software designers. The platform master was responsible for collecting all information on updates and changes made to the software, for ensuring that the software was tested and corrected, and for ensuring that the project schedule was updated with revised plans and activities before the platform was released.

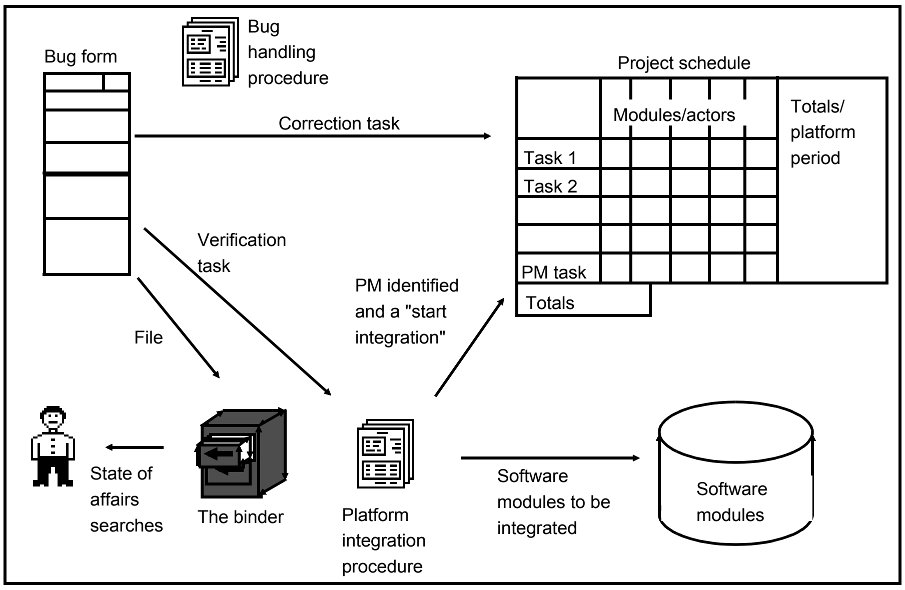
In the S4000 project one of the spec-team members was responsible for the overall plans. He was responsible for incorporating the new tasks (e.g., correction tasks), changes, etc. into the plans. The plans were organized in a large spread-sheet containing information on: tasks to be accomplished and references to detailed descriptions of tasks, estimated amount of labor-time per module for each task, responsibility relationships between software modules and software designers, relationships between tasks and platform periods, and total planned work hours per platform period for each software designer.

In order to coordinate the activities of handling bugs in the project a multitude of discussions, ad hoc meetings, and planned meetings were conducted throughout the project. In relation to the registration of a bug, the testers engaged in discussions of the problems they had identified, or they discussed the classification of a bug with one of the designers. The process of diagnosing bugs was organized as a structured meeting once a week where the spec-team (and sometimes other involved designers) discussed and negotiated the diagnosis and classification of the reported bugs, and how to incorporate the correction into the plans. Based on our observations, one out of four bug reports required discussion and negotiation between a tester and a designer, or between the spec-team and a designer. A spec-team member typically spent one day a week diagnosing bugs and negotiating bug classification and resource allocation with testers and software designers. When correcting bugs the designers often engaged in ad hoc discussions with other designers and in negotiations with, for example, the marketing people (who were responsible for the overall requirements) of the actions that would be acceptable. Verification was done during the integration periods. The designers spent much time during the integration periods negotiating acceptable solutions to the problems that had occurred. The integration process also contained a structured meeting in which all software designers participated. At this meeting all problems and solutions were presented. Our observations indicate that during integration, all of the software designers spent half their time coordinating the software integration and negotiating how to deal with the problems at hand.

As outlined above, coordination activities were also supported by means of forms, lists, procedures, etc.: (1) The bug form and the related procedure stipulated the flow of the bug registration and correction work; (2) the spread-sheet provided a conceptual structure for scheduling tasks, actors, and deadlines by relating development activities to relevant software modules and to responsible actors; and (3) the platform periods established a common basis for the designers' activities, thus facilitating overview of the state of affairs, guaranteeing that the software components could be integrated, and that the corrections conducted were verified. Figure 3 provides an overview of the artifacts involved in the coordination of software testing. The invention, implementation, refinement, and use of new means supporting the coordination evolved fast and without serious complications. A small group of people organized the work and several of the forms, lists, etc. were invented due to needs realized by the group itself.

The characteristics described above are in many respects similar to what can be observed in office work, e.g., the ac-

**FIGURE 3. Artifacts and prescriptions supporting the coordination of the bug handling process. The registered bugs create new tasks which are incorporated in the project plan. The project plan defines when the next platform integration must be started. The bug forms provide input for the platform integration by specifying verification tasks. The binder and the procedure for handling bug forms ensure that all bugs are treated, and in addition they provide an overview of state of affairs in the process**



tivities and roles (Hirschheim 1985), although they mainly address the coordination aspects of the bug registration and correction process. The complexity of the actual cooperative work, e.g., the problems of identifying the cause of a bug, will not be addressed in this paper.

## 5. Identifying Needs for Coordination Support

The software testing process in the S4000 project at Foss Electric contained a large number of activities. In this section we present and discuss coordination work related to the activities of: registration, classification, diagnosis, correction, verification, and monitoring the state of affairs. For each of these activities we furthermore discuss the functionality supporting the coordination which could be provided by computer technology. In order to structure the discussion, these activities are presented as distinct stages of work, and for each stage the analysis of the work and the discussion of need for computer support are presented in two separate subsections. Section 5.1 discusses support of the work flow, and the following sections discuss each of the stages of work.

### *5.1. Supporting the Work Flow of Software Testing*

*Work:* In the S4000 project, registration, classification, and correction of bugs were distributed activities. Diagnosis of software errors was done by the spec-team at a weekly meeting. This meeting resulted in correction requests being distributed to the 5–10 software designers responsible for correcting the bugs. The platform master was, during the integration period, responsible for verifying the corrections. As an example, a marketing person tested the usability of the user interface and realized that some of functionality was not accessible from the menu structure. The problem was then classified and described in a bug form and sent to the spec-team. They decided that Hans, as responsible for the UI-module, should make a correction. After Hans had completed the correction he reported to Jens, who was platform master for the next integration period, that the problem had been dealt with. During the platform integration week Jens tested and accepted the corrections. In summary, this process was organized as distributed testing, centralized diagnosis, distributed correction, and centralized verification.

*Support:* A work flow of distributed error registration, classification, diagnosis, correction, and verification, among other things, needs support for routing information between actors. When an actor has completed his or her activities in relation to a specific bug form, the work flow system must automatically validate that the required information is registered, then pass the information on to the next actor (or group of actors), and finally notify the receiver(s) that new action must be taken. However, in most situations is it impossible to completely specify all situations which may occur (see e.g., Suchman 1987, Schmidt 1994). The coordination of software testing at Foss Electric was certainly no exception to this. Thus, the actor completing an activity must be able to overrule the routing and redirect the information to somebody else. The protocol stipulating the routing must furthermore be based on roles to which actors can be related. The study at Foss Electric clearly illustrates that the actors had several roles, and, more importantly, that these roles were relatively stable entities in the work setting. Some roles were played by different actors at different points in time, e.g., the role of platform master. Changing the actor related to a role should not imply changes to the protocol stipulating the routing. Monitoring the progress of the work is essential when coordinating work. This requires a consistent and updated data-base containing information on all reported bugs and their current status. Negotiation of classifications, diagnoses, allocation of resources, deadlines, etc. were a predominant feature of the software testing work. Approximately one out of five of the correction tasks defined by the spec-team resulted in negotiation between the spec-team and the responsible designer. Support for actors engaging in negotiating the bug handling process could draw upon research addressing the support for negotiation (Flores *et al.* 1988), or available work flow technologies (e.g., De Cindio *et al.* 1988). It is beyond the scope of this paper to review this further.

## *5.2. Registration and Classification*

*Work:* The S4000 software was tested on software simulators and on instrument prototypes, and the project had distributed detection, registration, and classification of software bugs. Occasionally testers and software designers engaged themselves in preliminary discussions on the interpretation of problems or possible bugs. Some errors were impossible to reproduce and hence difficult to describe during registration. Our observations indicate that in at least 20% of the bug reports, testers were not able to describe the problem in detail. If it was difficult for the tester to fill in the required information in the bug form, the spec-team member filled in the form after having discussed the problem with the tester. As one of the spec-team members said:

> "The form is not very user-friendly. We often have to force them *[the testers]* to fill in a form. Sometime they just send in a note describing what they have seen, and we must produce a form."

The classification of errors as either catastrophic, essential or cosmetic was done according to the tester's perspective. As one of the spec-team members phrased it:

> "People, depending on who they are, often interprets a catastrophe in a different way than I do. An inconsistency in the user interface might, for example, be a disaster to a marketing guy, whereas it is a cosmetic problem to me".

Some of the testers tried to maintain awareness of errors identified by other testers in order to obtain useful information for a first diagnosis of the problem. This was, however, a difficult task. If, for example, a tester from the marketing department wanted to check recent problems with the menu structure, this could imply browsing through more than 500 forms in the binder.

*Support:* The bug handling system studied could be improved by refining the bug classification system. Too often the existing classification structure led to discussions resulting in the spec-team reclassifying bugs. A more elaborated classification of the type and importance of bugs could support the testers in providing useful information to the spec-team and to designers. We suggests two classification structures: A classification of the phenomena observed (program stopped, window in wrong place, unstable output on tests, etc.) and a two-dimensional classification structure reflecting software quality parameters (maintainability, marketing, stability, safety, usability etc.) and the testers assessment of the level of importance. Research within software engineering may provide input for more elaborate classification structures, for example, standard software quality taxonomies (Boehm 1981, Fairley 1985. If the classification structures have a "miscellaneous" category on each level, the software tester is provided with an opportunity to classify in the most unambiguous way. These "other" fields might contain text annotations, allowing testers to further specify and characterize the problem. Support for filling in bug forms could be supported through a facility for retrieving registrations of similar bugs based on more elaborate classification structures and using a central database containing information on all bugs. Classification structures reduces complexity of coordination work by providing a conceptual structure that makes it possible for testers and designers to perform distributed storing and retrieval of bug forms with a minimum of peer-to-peer coordination. Also, support for discussions via electronic mail or bulletin boards among testers could improve the quality of the information registered.

### *5.3. Diagnosis*

*Work:* Diagnoses were mainly done by the spec-team members at a weekly meeting—more frequently in periods. They browsed the submitted bug forms. The number of forms varied a lot from week to week—in the last half a year of the project it was around 25–30 forms per week. The status and classification of each bug were discussed. If the classification differed very much from the one made by the tester, the team sometimes summoned the tester and discussed the classification with him. If two registered bugs were diagnosed as being identical, only one of the forms was processed further. The spec-team then discussed the diagnosis and responsibility by reviewing specifications, documentation, source code, etc. In the complicated cases (about 10%) the team summoned the designers responsible for the relevant modules in order to negotiate the diagnosis, the responsibility, and an estimated correction time. Responsibilities and time-estimates for corrections were incorporated as tasks in the planning spread-sheet. The allocation of correction tasks to platform periods was decided based on assessments of the workload of the responsible designer(s) and of the importance of the problem. The spec-team either handed over or sent bug forms to the designers. This could result in discussions among designers and spec-team members about the diagnoses and time-estimates.

*Support:* Supporting diagnosis of bugs primarily implies supporting communication among the spec-team members. The field study clearly illustrates, that without face-to-face communication, the spec-team members would have had severe problems. E-mail based communication between spec-team members, testers and designers might, however, support diagnoses and prevent some of the ad hoc discussions. The coordination of the diagnosis work could also have been supported by providing access to information on already reported bugs. As discussed in the previous section, improved classification structures could provide better support for diagnosing bugs by more clearly stipulating testers' assessment of type and importance of the bug. The task of meshing new correction tasks and existing plans is quite complicated. This could be supported by providing access to information on: relationships between roles and actors, architecture of the software complex, relationships between software modules and responsible designers, designers' workloads, existing work plans, and relationships between tasks and deadlines, etc. The needs for obtaining an overview of the existing bugs and plans will be discussed further in Section 5.6.

### *5.4. Correction*

*Work:* Bugs were corrected in a distributed manner. Although the structure of the bug form did not support the allocation of responsibility for correcting one particular bug to several designers, this often happened. Often a designer discovered that the problem he or she was correcting affected modules owned by other designers, thus creating leading to a need for coordination of who was going to do what, when and how. Since all the 5 to 10 designers making corrections were placed in the same room, this coordination was mainly conducted by an abundance of meetings and discussions. This, however, imposed a problem when the complexity of the software increased. One of the designers characterized the problem as follows:

> "The problem we have right now is that the software architecture is difficult to decompose so much that one designer can handle a component. We are all working on several components, and work on a single component involves two to four men, and perhaps even some of the electronical designers too. Then we need coordination [..] We have recently started a process where we try to produce more formal documents and agreements about the things we work with, we haven't been good at doing this before, but now we have to do it."

The first thing a designer usually did upon receiving a request for correcting a bug was to consider both the diagnosis, the estimated correction time and the deadline—the platform period. If the diagnosis or estimate did not seem acceptable, the designer contacted one of the spec-team members in order to negotiate the situation. With respect to the estimates, the spec-team had a policy stating that the designer was always right. When requirements could not be met within the scheduled time limit, this led to negotiations with the marketing people responsible for the requirements specification.

*Support:* The most obvious support for coordinating software correction activities is by providing good and accessible means for communication. This was to a certain extent already facilitated by placing all designers in the same room but electronic communication systems might be of help as well. Furthermore, support for the process of requesting and rejecting correction tasks must be provided, i.e., if coordination is supported by a work flow system, the designers must be able to reject a request—return it to the originator with a comment—or pass the request on to another designer pointed out to be the one that should have been assigned to the task. Improved formalization and structuring of the specification of modules, module interfaces, message handling, etc. could also support the coordination (Parnas 1985). In the S4000 project interactions among modules were only very loosely sketched, thus resulting in designers engaging in a ad hoc coordination. Improved use of some of the existing specification techniques or CASE tools could decrease the need for ad hoc coordination by providing an improved structuring of the field of work, i.e., the software system being designed (Mathiassen & Sørensen 1994). The designers pointed at the problem of being aware of the changes other designers made, or to be able to ensure that all the others were aware of a correction, idea, or problem. Improved support for the documentation of corrections could, for example, be provided by classification structures and browsing facilities for the database containing information on the bugs and corrections.

### *5.5. Verification*

*Work:* In the platform integration process, initiated and managed by the platform master, all modules were linked and compiled, and the software was tested prior to its release, all corrections reported were verified, tasks identified during integration and testing were meshed into the plans, and designers were informed about the state of affairs. During the platform period all designers worked on testing the integrated software and were constantly meeting and discussing the results and problems of the integration. The platform master coordinated the activities and delegated subtasks to the designers. As a designer put it:

> "Usually we produce a list of all the problems we have identified on a large white board. We then discuss whether this is a problem—an error—or not. Actually it's the platform master who does that. If it's a real heavy problem you are immediately summoned and asked to correct it."

A set of brief organizational procedures stated how to organize the integration process, and contained a number of check lists and standards covering the details to be checked before the platform master could release the software for further modification. Only one pre-scheduled and structured meeting was held at the end of the integration period. All designers had to participate in this meeting where they in turn described the changes they had implemented since the last integration period.

*Support:* Close interaction among software designers during verification is essential. At Foss Electric this was solved by placing the designers in the same room. In cases involving of a large number of project members or geographically distributed development this solution is not feasible. Here a work flow structure supporting the division of labor, a structure for establishing a common language (e.g., classification structures, module specifications, etc.) and electronic meeting and communication systems could provide support. In the coordination of concrete verification tasks, support for distributing responsibilities should be provided. In the S4000 project this was handled by the platform master who personally delegated each correction task. It would be obvious to support this by having a computer supported procedure for delegating the verification tasks, and support in reporting back. This, of course, would also include support for registering new bugs, similar to what was discussed in section 6.2. In order to improve the awareness of changes made by others and to establish a common understanding of the software complex, some support for "viewing" the structure of the software complex must be provided. From our observations it was obvious that the designers had problems in relating themselves to the structure of others designers' modules although these had an essential impact on how they should (re-)design their own modules. This could be supported by the production technology dimension of CASE tools (Henderson & Cooprider 1990).

### *5.6. Monitoring*

*Work:* A central activity in coordinating the process of registering and correcting bugs in the S4000 project was to establish an overview of the state of affairs, i.e., the progress of the process, the number of bugs that still needed to be corrected, the accumulated estimation of correction time, changes that might affect other modules, etc. The designers needed to be aware of corrections and changes affecting their modules. The spec-team members needed to know the state of affairs before each spec-team meeting. The testers frequently tried to obtain an overview in order to avoid wasting their time on reporting already registered bugs. The platform master needed an overview of corrections to be verified in the next integration period in order to plan the integration work. Management tried to get an overview of the progress of the whole development project. There were basically three sources for this type of information: informal communication, the bug form binder, and the list of bugs which had not been corrected. There was a lot of informal communication and discussion among the designers about what kind of corrections and changes they had made. Some of the testers discussed changes in the software with the designers several times a week, whereas others never contacted the designers directly. Even though the designers sat in the same room and were engaged in discussions every day, it was difficult for them to be aware of the state of affairs:

> "Usually, the channel driver guy and I had a clear deal. Verbal discussions and a sketch drawing were sufficient. But in a project as large as the S4000 we don't have a complete overview of the software complex. And then you are in big trouble when the other guys change their code" (Software designer in the S4000 project).

Testers as well as designers found it very difficult to obtain the necessary overview by consulting the bug form binder, mainly because the forms were only organized according to the seven categories presented in Section 5.2. This made it almost impossible to determine whether the same bug had been reported in several bug forms. It was the intention that a specification of the corrections should be included in each form (cf. the bottom of the form in Figure 1). In order to be aware of relevant changes, the designers were expected to browse through all the forms in order to see if anything there was of interest. The binder contained approximately 1400 forms at the end of the project!

The third source for getting an overview was a weekly produced list of registered bugs that had not yet been dealt with. One of the designers phrased the problems with this as:

> "Originally the intention was to produce statistics of the number of known-but-not-yet-fixed problems and use this as a management tool. The management hoped to find a decreasing curve on the week-to-week measurement. They didn't. But we realized that as a management tool this can only be used if you have a stable product. We didn't have that."

*Support:* Obtaining awareness of the state of affairs by monitoring progress of the software testing process plays an important role as a fundament for coordinating activities, and this should be supported by several means. More elaborate classification structures for bugs (cf. Section 5.2), for the software modules and their interaction (cf. Section 5.5) and for the relationships between actors, roles and resources could facilitate assessments of the relative importance of these issues. Providing designers and testers with browsing and query facilities to a database containing all registered bugs would enable these actors to access aggregated information on reported bugs which have not yet been corrected, the number of a specific type of bugs, the number of not yet corrected bugs in a specific module, the number of bugs a specific designer is responsible for getting corrected, etc. Also access to view the project schedule would be useful for monitoring state of affairs. This functionality could be provided by means of some of the existing project planning tools (e.g., Microsoft Project), and should support requests like: Who is responsible for the UIS-module? Which modules are John responsible for? How busy is Tom the next integration period? How much time has been spent on correction so far? What percentage of the corrections does this correspond to? etc.

## 6. Discussion

We have, in this article, analyzed the coordination of distributed software testing of one project in one organization. We have focused on coordination work related to distributed actors performing registration, classification, diagnosis, correction and verification of software bugs applying a bug form work flow system, a resource-planning spread-sheet, and a configuration control procedure. Based on this analysis we have discussed needs for computer supporting this coordination work. The project we have looked at is most likely not unique. Many software projects faced with the immense complexity of distributed software testing use various kinds of forms, classification structures and organizational procedures in order to cope with the complexity of coordinating of software testing activities.

Kraut and Streeter (1995) argue that computer support of the informal and direct communication in systems development is required, and they suggest tools supporting conferences and distributed meetings. The analysis in this paper can be viewed as work elaborating on Kraut & Streeters conclusion that informal communication is very important in systems development, but faced with the need for coordinating an abundance of distributed decisions, there is a need for formal coordination means due to the excessive transaction costs and the ephemeral nature of informal communication.

Conceptual structures played a crucial role for the actors when coordinating the software testing activities. The software was divided up into a set op modules, which also were represented in the project schedule. The architecture of the system represented an aggregation of the software modules. Work plans represented structures of actors and roles involved, resources available, tasks to be performed. Software bugs were classified according to their importance and the bug form played an important role in creating and representing bugs as separate entities, i.e., the bug form was the instrument used for transforming observed phenomena to a set of identified software bugs. These structures provided the actors with an overview of both the field of work which was registration and correction of bugs in the S4000 software, as well as of the cooperative work arrangement, e.g., the actors, their roles, and the resources available. The structures can be viewed as dimensions along which coordination is conducted (Schmidt & Simone 1995), i.e., the coordination activities are performed by referring to abstractions and conceptualizations of the nature of the work, not by directly interacting with the objects of the work (e.g., the code). We have identified a set of actions performed by the actors in relation to these conceptual structures. They were *classifying* bugs, tasks, modules, etc., *routing* information and requests, *monitoring* state of affairs, *allocating* resources, *meshing* work products, and *negotiating* diagnoses, allocation of resources, etc. When discussing computer support of coordination aspects of software testing, it is obvious to require access to structures in the computer based system similar to the conceptual structures mentioned above, and to facilities supporting the basic actions listed.

In their functional model of design aid technology, Henderson & Cooprider (1990) argue that design aid environments mainly consist of three components: (1) production technology containing facility for representation, analysis and transformation of objects, relationships and processes; (2) coordination technology with control functionality supporting planning and enforcing rules, policies that will govern or restrict the design process and with cooperative functionality enabling users to exchange information relevant for the work process; and (3) organizational technology with support functionality to help users and with infrastructure functionality providing standards enabling portability of skills knowledge, procedures or methods across projects.

This article has presented an analysis of needs for computer support pertaining to the coordination technology component, and its relationships to the production technology component. CASE tools supporting the conceptualization of the software architecture, as well as test planning tools have been available for more than 10 years. These tools primarily provide functionality within the production technology component, i.e., support the individual test tasks, or provide an overview of the state of affairs by applying a specific set of testing metrics. Although certain coordination aspects are supported by these tools, several improvements are required. Furthermore, the coordination support must be integrated with existing tools and techniques, for example cooperative software inspection tools (Freedman & Weinberg 1990, Mashayekhi *et al.* 1993). Although we have begun an identification of such support needs, much work still remains to be done before the ideas can be realized.

Future work should go in at least three directions. One is to evaluate related products and concepts, i.e., relate the support of the cooperative aspects to the individual aspects of software testing and reflect on how cooperative work can be computer supported. Henderson and Cooprider's model will be relevant as a starting point for this work. The second direction is to tie our body of work to the ongoing efforts in conceptualizing relevant aspect of cooperative work, for example the concept of Coordination Mechanisms (Schmidt & Simone 1995) and the third direction will be to build experimental prototypes as a step forward in concretizing the ideas.

**Acknowledgments**
This research could not have been conducted without the invaluable help of numerous people at Foss Electric. Thanks to Leif Løvborg and three anonymous reviewers for many useful comments and suggestions, and to Susan Leigh Star for helping us understanding central aspects of classification schemes. We also thank Finn Kensing, Betty Hewitt, and IRIS'94 reviewers for constructive comments and ideas to previous versions of this paper. This research is partially funded by the Esprit BRA 6225 COMIC project, the Finnish Center for Technology Development (TEKES), and the Danish

Technical Research Council project CODEM sponsored by Fisker og Nielsens Fond, and Ib Henriksens Fond. All errors in this paper naturally remain the responsibility of the authors.

## References

Bannon, L., (1993). CSCW: An Initial Exploration. *Scandinavian Journal of Information Systems*, (5): 3-24.

Beizer, B., (1990). *Software Testing Techniques*. Second Edition. Van Nostrand Reinhold.

Boehm, B. W., (1981). *Software Engineering Economics*. Prentice-Hall.

Brooks Jr., F. P., (1982). *The Mythical Man-Month — Essays on Software Engineering*. Addison-Wesley.

Bucciarelli, L. L., (1984). Reflective practice in engineering design. *Design Studies*, 5(3): 185-190.

Carstensen, P., (1994). The Bug Report Form. In K. Schmidt ed. *Social Mechanisms of Interaction*. pages 185-216, Esprit BRA 6225 COMIC

Carstensen, P., C. Sørensen & H. Borstrøm, (1995). Two is Fine, Four is a Mess — Reducing Complexity of Articulation Work in Manufacturing. In *COOP'95. Proceedings of the International Workshop on the Design of Cooperative Systems, January 25-27, Antibes-Juan-les-Pins, France*, INRIA, Sophia Antipolis, pages 314-333.

Dahlbom, B. & L. Mathiassen, (1993). *Computers in Context — The Philosophy and Practice of Systems Design*. Blackwell Publishers.

Dalal, S. R., J. R. Hogan & J. R. Kettering, (1993). Reliable Software and Communication: Software Quality, Reliability, and Safety. In *15th International Conference on Software Engineering, Baltimore, Maryland, USA*, IEEE Computer Society Press, Los Alamitos, California, USA, pages 425-435.

De Cindio, F., C. Simone, R. Vassallo & A. Zanaboni, (1988). CHAOS: a knowledge-based system for conversing within offices. In W. Lamersdorf ed. *Office Knowledge: Representation, Management and Utilization*. Elsevier Science Publishers B.V., North -Holland.

Dijkstra, E. W., (1989). On the cruelty of really teaching computer science. *Communications of the ACM*, 32(12): 1398-1404.

Fairley, R., (1985). *Software Engineering Concepts*. McGraw-Hill.

Flores, F., M. Graves, B. Hartfield & T. Winograd, (1988). Computer Systems and the Design of Organizational Interaction. *TOIS*, 6(2): 153-172.

Freedman, D. & G. Weinberg, (1990). *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Dorset House.

Gelperin, D. & B. Hetzel, (1988). The Growth of Software Testing. *Communications of the ACM*, 31(6): 687-695.

Hamlet, R., (1988). Special section on software testing. *Communications of the ACM*, 31(6): 662-667.

Henderson, J. C. & J. G. Cooprider, (1990). Dimensions of I/S Planning and Design Aids: A Functional Model of CASE Technology. *Information Systems Research*, 1(3): 227-254.

Hetzel, B., (1988). *The Complete Guide to Software Testing*. QED Information Sciences Inc.

Hirschheim, R. A., (1985). *Office Automation: A Social and Organizational Perspective*. Information Systems Series, eds. R. Boland & R. Hirschheim. John Wiley and Sons.

Jennings, K., (1990). *The Devouring Fungus*. W.W. Norton & Company.

Johnson, P. M. & D. Tjahjono, (1993). Improving Software Quality through Computer Supported Collaborative Review. In G. De Michelis, C. Simone & K. Schmidt eds. *ECSCW '93. Proceedings* *of the Third European Conference on Computer-Supported Cooperative Work, 13-17 September 1993, Milan, Italy.* pages 61-76, Kluwer Academic Publishers.

Kraut, R. E. & L. A. Streeter, (1995). Coordination in Software Development. *Communications of the ACM*, 38(3): 69–81.

Mashayekhi, V., J. M. Drake, W.-T. Tsai & J. Riedl, (1993). Distributed Collaborative Software Inspection. *IEEE Software*, (September): 66–75.

Mason, R. O., (1989). MIS Experiments: A Pragmatic Perspective. In I. Benbasat ed. *The Information Systems Research Challenge: Experimental Research Methods*,vol. 2. pages 3-20, Harvard Business School Research Colloquium, Harvard Business School

Mathiassen, L. & C. Sørensen, (1994). Managing CASE Introduction — Beyond Software Process Maturity. In J. W. Ross ed. *Proceedings of the 1994 ACM SIGCPR Conference, Old Town Alexandria, Virginia, USA*, ACM, pages 242–251.

Myers, G. J., (1979). *The Art of Software Testing*. John Wiley and Sons.

Parnas, D. L., (1985). Software Aspects of Strategic Defence Systems. *Communications of the ACM*, 28(12): 1326–1335.

Patton, M. Q., (1980). *Qualitative Evaluation Methods*. Sage Publications.

Petchenik, N. H., (1985). Practical Priorities in System Testing. *IEEE Software*, 2(5): 18-23.

Pressman, R. S., (1988). *Making Software Engineering Happen - A Guide for Instituting the Technology.* Prentice-Hall.

Schmidt, K., (1991). Riding a Tiger, or Computer Supported Cooperative Work. In L. Bannon, M. Robinson & K. Schmidt eds. *ECSCW '91. Proceedings of the Second European Conference on Computer-Supported Cooperative Work*. pages 1-16, Kluwer Academic Publishers

Schmidt, K., (1994). *Modes and Mechanisms of Interaction in Cooperative Work.* Risø National Laboratory, [Risø-R-666].

Schmidt, K. & P. Carstensen, (1990). *Arbejdsanalyse. Teori og praksis [Work Analysis. Theory and Practice]*. Risø National Laboratory, [Risø-M-2889].

Schmidt, K. & C. Simone, (1995). Mechanisms of Interaction: An Approach to CSCW Systems Design. In *COOP'95. Proceedings of the International Workshop on the Design of Cooperative Systems, January 25-27, Antibes-Juan-les-Pins, France*, INRIA, Sophia Antipolis, pages 56-75.

Siemieniuch, C., (1992). Design to product — A prototype of a system to enable design for manufacturability. In M. Helander & M. Nagamachi eds. *Design for Manufacturability — A Systems Approach to Concurrent Engineering and Ergonomics*. pages 35-54, Taylor & Francis.

Sørensen, C., P. Carstensen & H. Borstrøm, (1994). We Can't Go On Meeting Like This! Artifacts Making it Easier to Work Together in Manufacturing. In S. Howard & Y. K. Leung eds. *Harmony Through Working Together, OZCHI 1994, Melbourne, Australia*, CHISIG, pages 181–186.

Suchman, L. A., (1987). *Plans and situated actions. The problem of human-machine communication*. Cambridge University Press.

Swenson, K. D., R. J. Maxwell, T. Matsumoto, B. Saghari & K. Irwin, (1994). A business process environment supporting collaborative planning. *Collaborative Computing*, (1): 15-34.

Yin, R. K., (1989). Research Design Issues in Using the Case Study Method to Study Management Information Systems. In J. I. Cash & P. R. Lawrence eds. *The Information Systems Research Challenge: Qualitative Research Methods*,vol. 1. pages 1-6, Harvard Business School Research Colloquium, Harvard Business School

Yourdon, E., (1988). Software Quality Assurance in the 1990s. In *Sixth Annual Pacific*

*Northwest Software Quality Conference, Portland, Oregon,*