Visual Support for Reengineering Work Processes

Keith D. Swenson

Collaboration Software, Program Products Division
Fujitsu Open Systems Solutions, Inc.
3055 Orchard Dr. San Jose, CA, 95134
+1 408 456-7667 kswenson@ossi.com

ABSTRACT

A model for collaborative work process and a graphical language to support this model is presented. The model allows for informal flow of communications and flexible access to information along with a formal flow of responsibility. Work is decomposed into a network of task assignments (actually requests for those tasks), which may be recursively decomposed to finer grained tasks. The model includes consideration for authority and responsibility. Process flow can be dynamically modified. Policies (templates for a process) may be tailored to provide versions of a process customized for different individuals. The visual language is designed to ease the creation of policies and modification of ongoing processes, as well as to display the status of an active process.

KEYWORDS

Visual language, collaboration, work flow, process modeling, business process reengineering.

INTRODUCTION

Information technology has proven to be a great aid in automating and accelerating well defined production activities. While computers have become relatively ubiquitous in the office setting, there is a lack of conclusive evidence suggesting that computers have been successful in increasing productivity of an organization at an aggregate level. There are undoubtedly a number of reasons for this "Productivity Paradox".[18][2][26][15]

One very promising approach to increasing organizational productivity through the use of information technology is termed "Business Process Reengineering"[30][3]. Studies have shown that automating an existing manual work process will have a very slight effect on productivity[18]. Instead, if the entire process is examined, and then redesigned to take into account capabilities provided by information technology, phenomenal increases in productivity can be achieved[9]. These increases in productivity typically arise out of reducing the number of individual steps to complete a process[8].

COOCS'93 - 11/93/CA, USA
© 1993 ACM 0-89791-627-1/93/0010...$1.50

The new process is made out of tasks which are less menial in nature, and the individual is empowered to play a more influential role in the process. Thus the advantages of BPR are not only those of improving organizational productivity, but also as a way to improve individual "quality of work life".

Though the benefits of BPR are known, organizations are finding it exceedingly difficult to implement. Existing software (mainly workflow packages) are too inflexible to fit the needs of all of the different users and groups within an organization. The actual processes within an organization are largely unknown to any single individual,

Regatta Project Goals

The Regatta group was formed in 1991 in order to develop software to support workgroups and to aid in reengineering work processes[28][29]. In order to achieve this the Regatta project has set three specific goals:

1. To make software that supports the *coordination* of activities of different members of a group by automating work processes.

2. To make software that allows every individual in a group to gain a better *understanding* of how their group or organization works. It is not sufficient that a work flow tool act as a black box process. It is furthermore not sufficient for the tool to assume that the present process is known, it must help in the discovery of the process.

3. To make software that supports the *change* of processes, by explaining the process to users.

Requirements for the Model

Ease of Process Definition: Collaborative processes need to be defined in such a way that the average user can change and create process descriptions. Experience with non-programmers suggests the use of a graphical approach to describing a process.[25][12]

Ease of Monitoring Process: In addition to being easy to program, one purpose of the graphical representation is to visually represent the current state of a process to help the participants in the process understand what has happened and what may happen next.

Dynamic Modification: Once a process has begun execution, it may be changed due to special circumstances not foreseeable at the time of definition. A plan (the definition of a process) must be modified easily on a case by case basis, without requiring that the process be restarted at the beginning. Requiring that the process be complete to the n-th detail before starting would be a barrier to use in most situations.

Partial Definitions: It is important that the user is able to define part of the plan in order to start the process, and is able to add to the plan description as the process advances. It has been suggested that it may not be possible to completely define many processes beforehand due to inconsistent micro-theories of how the task should be done whose conflicts do not get resolved until after the process is started.[11]

Individually Tailorable: The model must be designed such that each person or group can supply their own policy for a particular task or goal. A policy is a template that can automatically create a plan for a particular situation. The system must maintain multiple versions of a policy, potentially one for each person or group. It must "inherit" a group or organization version of the policy, when no individual policy exists. When an individual receives a request, the system must use the appropriate policy.

Abstraction and Decomposition: Since each person views what is happening in their own way, it is important that the plan they define works at the level that they do. Higher level people use higher level generalizations and "manipulate" at a more abstract level, while lower level people are concerned with the details.

Control of plan: Since the system allows the person who is responsible for a particular result to modify that part of the plan, it must protect that part from modification by others. Thus the system must support the idea that different parts of the process are controlled by different participants in the process. This assures that the plan belonging to the responsible person is being followed.

Authority: Since steps in a process are real and potentially lengthy tasks, the system must be clear about the authority of the origination of the tasks. Any user may create a process, but it should not be possible for a user to create a task for the accounting department to produce a large check without following proper procedures. To do this, the system should accurately communicate the authority of the originator of a request.

Negotiation: While the system automates the assignments of tasks to people, it is important that users retain the ability to manage their own work load. An assignee too busy or otherwise unable to handle a task may decline the task, and the system must raise some form of an exception. This mechanism should support a rich free-form communication between the requester and the assignee within the context of the process so that the task might be reassigned, or the process modified so that work can continue. In a complex system such exceptions might be common, and it must be a feature of the system to support the resolution of such conflicts.

Delegation: A plan must be able to represent delegation well, so that when one person delegates a task to another, it appears as a delegated task, and not as a myriad of detailed things for the other person to perform.

Automation: Users need to understand not only that the task exists to be performed, but also some idea of how the task came to be assigned to them. The system must automate process so that workers can concentrate upon their specialization, and spend less time on coordination issues.

Open: The system needs to support the use of tools that are specialized to the user's needs, and not to require that all work be completed within the system. The model should anticipate use of external tools and still provide sequencing support of the various tasks that need to be done with those tools.

THE MODEL

The model supplies a variety of ways to support the communications needed to coordinate activities within a group.

Colloquies

Primarily, the model provides a shared collaboration space, called a *colloquy*, in which to coordinate a set of tasks that are performed to accomplish a specified goal. Much has been said about the benefit of such a shared space for collaboration.[21][20] ConversationBuilder[13][14] has shown that such an approach can be an effective framework for a wide variety of collaborative activities. Like ConversationBuilder, this model provides for active support of collaboration.

A colloquy is composed of a shared data space and a collection of plans. The plans are composed of stages and roles. Stages each contain a collection of actions. The shared data space can hold documents and other artifacts.

The colloquy also defines the group of people who may have access to the information within the colloquy, thereby providing a measure of control of the distribution of information.

Actions & Messages

Users act on the colloquy through actions which are either built-in or user defined. It is important to remember that the main design goal is to facilitate communications between the users of the system. Therefore, with every action users may include a message with details about what they have done, or why they have made a particular choice. This is particularly important if a non-standard action is chosen, such as a rejection. The message explaining why a choice was made, or what needs to be done before a particular action will be taken, is appropriately kept within the context of the colloquy.

History
A record of all actions and their associated messages can be browsed at any time by any participant of the colloquy. This is a way for a new participant of the colloquy to catch up on what has happened up to this point. Since date and time are automatically recorded for each action it is easy to determine which tasks are taking the longest and might possibly need some fine tuning of the process.

Plans
The process is modelled as requests for tasks. Plans are composed of a network of stages, each stage representing a task request, commitment, or question as a specific step in the process. Each stage may be active or inactive indicating whether the associated task is currently under way.

A stage includes one or more user defined actions, called *options*. Each option (with it's associated message) represents a declaration that the assignee may make to represent the results of the task or decision. The chosen option causes an *event* to be sent which ultimately activates and deactivates other stages. Any number of stages may be active at the same time, allowing work to proceed in parallel.

The person who is responsible for the result of the plan is the *owner* of the plan. The plan represents the owner's point of view. The owner is usually the creator of the plan, and is the only person who may make changes in the plan.

Plans are created and modified by using a graphical representation called *Visual Process Language* (VPL). As the process is enacted, the current status of the process is viewed with the same graphical representation.

Stages
Strictly speaking stages are not tasks, but rather the communications needed to coordinate tasks. A stage is always a request from one person (the plan owner) to another person (the assignee). In Searle's taxonomy of speech acts[22][23] the stages represent either directives (requests to another person) or commissive (if assigned to yourself). Stages appear in the VPL plans as ellipses.

The key difference between a task and a request for a task is that a request may be *accepted* or *declined*. A request is accepted to communicate that the task will be done. Declining a request is really a message back to the plan owner that something is wrong. The owner may then choose to assign the request to someone else, or may choose to change the request to make it acceptable. In a real world situation the owner and assignee may negotiate back and forth several times before coming to agreement on the task to be done. The user need not plan for such a dialog because it is built into the collaboration model.

The request may be expressed in any amount of detail; it is not constrained to a set of predefined tasks. This allows language to be used naturally for the precise description of the task, and represents the Regatta philosophy of supporting communications, not restricting it.

Roles
Each stage has an assigned *role*, which appears in the upper part of the stage's ellipse. A role is simply a container for a list of names of people or groups. A role is not a quality of an individual, but rather a relationship between a person (or group) and a particular colloquy. A given person may play one, two, or all of the roles simultaneously in one colloquy, while playing different roles in another colloquy. The role that is assigned to the stage is said to be responsible for the stage.

Forms
Each stage has a form which is used to display colloquy information at the time that stage is active. While many processes will use the same form for all stages, there are cases where a special view of the information is desired for specific stages of the process.

Options
From Searle's taxonomy of speech acts, a declaration is an utterance which in the act of speaking it changes the state of a group. Clearly, declarations are the key to coordinating the activities of the group. The options on a stage represent the declarations that the assignee may make as a result of performing the task. Small circles on the edge of a stage together with the arrow protruding from it represent an option. The option is labelled with a letter or a few letters to indicate the event that is expected to trigger the action. The actual event may have a longer name. In most cases the option will *complete* the stage and cause it to deactivate. Alternately, options can be set to leave the stage active after sending its event. This is indicated in VPL by using inverse coloration of the option.

The declaration may include any amount of detail necessary to communicate the result of the task. This detail is included in the message that can accompany any user action.

Each option appears as a menu item to the responsible person. When the task is completed manually, the assignee simply selects the menu item. Like a declaration, the act of choosing an option changes the state of the process. It does this by sending events to activate or terminate other stages.

Though events are represented as arrows from one stage to another stage, they should not be viewed as carrying data from one task to another task. The event is instead an abstract mechanism that is used purely to coordinate stages. The information output by a task or needed by a task is distributed to all active stages simultaneously by the colloquy mechanism.

Non-stage Nodes
While stages represent the major steps in a process, there are other kinds of nodes which provide some automated capabilities within the plan. Programmed nodes, represented as a small circle, can execute a user defined script when activated. Condition nodes send an event depending upon whether an expression evaluates to true or false. Timer nodes send events a specified number of days or weeks after being activated, or at a specified time in the future. Start nodes

(hexagons with "start" in them) are automatically activated when the plan is initialized. Exit nodes are used to send events from a subplan to its parent plan. Exit nodes are represented as hexagons labelled with the event they will send to the parent.

When two or more event arrows are pointing into a stage it means that the first activate event from any one of them will activate the stage. This OR-like behavior is common for all VPL nodes except for the AND-node which has the property that it receives all expected events before it sends any event. The AND-node is represented by a small circle with a "plus" symbol in it.

All non-stage nodes can send any number of events upon completion. This allows any non-stage node to start parallel tracks in a plan.

Split Stage

A split stage is a special kind of stage that makes a copy of itself for ever person assigned to its role. This allows each person in the list to respond uniquely and in any order to the request. An AND-node placed after a parallel split will cause the process to wait until the last person takes their action before continuing.

SubPlans

If the request is not to be completed manually, the assignee may create a subplan to accomplish the task. The assignee becomes the owner and creator of the new subplan and may make requests to others by creating stages within the subplan.

The subplan is ended by activating an exit node, when sends an event to the parent stage. If the event sent by the exit node matches an option on the original stage, then that option is triggered just as if the assignee had selected the corresponding menu item.

Subplan provide more than just hierarchical decomposition of tasks into subtasks. Since the subplan is owned and can be modified by a different person than the parent plan, it allows people to collaborate in the specification of the planning. The result is a collaborative planning tool.

Built-In Actions

In order to simplify the construction of meaningful processes, certain common actions and interactions are built into the stage. They help facilitate the process of coming to agreement about a particular task or question, and help to resolve exceptions.

The stage starts out inactive in a state called unstarted. Upon receipt of an activate event the stage becomes offered. When offered, it is the responsibility of the assignee and appears on the assignee's task list. The built-in actions *decline* causes the stage to become unassigned, which is the responsibility of the owner. The owner may then change the request or change the assignee and by choosing the action *reassign* but the stage back into the offered state. The *accept* action will but the stage in the manual state. Choosing *sub-plan* will create a subplan and put the stage in automatic mode.

These states work a little differently when the stage is offered to a group. An individual accepting the task has the effect of removing the rest of the group from the responsibles list, thereby reserving the task to himself. Then, after acceptance, that individual's policy might be used to create a sub-plan. Finally, the decline option for a group causes that individual to be removed from the list of names in the role. Only when everyone has declined the task, and the role is empty, does it go to the unassigned state thereby signalling the owner to get involved. An approximate state chart for a stage is shown in Figure 1. See [10] for a thorough treatment of state charts.

Finding Your Tasks

Assigning an active stage to a user is not complete until that user has been made aware that the task exists. The pending list functions as a to-do list, listing all of the colloquies within which that users currently has a pending action. As new requests are made to an individual, they appear on their task list. As actions are taken that complete stages, they disappear from the list. When a task is automated by a subplan, it disappears from the list.

Scripts

The Visual Process Language is a coordination language. Its capabilities are orthogonal to the procedural language needed to automate simple tasks. For this reason the interpreted scripting language called TCL (Tool Command Language)[19] is included. TCL can be extended by user defined commands and procedures. External programs can be called.

The purpose of a programmed node is to provide a place to put a user defined script that is executed when the node is activated. Stages has scripts that are evaluated when the stage is activated and when deactivated. Options placed on a stage have two scripts, one that is evaluated on the client side and one evaluated on the server side.

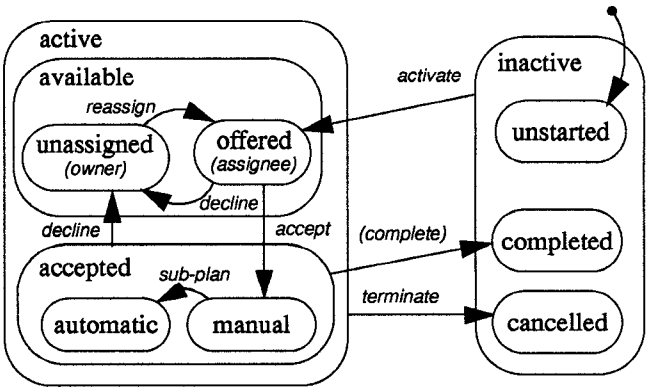

Figure 1. State chart diagram for a stage

EXAMPLE PLAN

An example plan which is used within a software development team to handle and process bug report is shown in Figure 2. It represents a Project Manager point of view on the way that a software development bug report is handled. This plan was invoked as a request from the submitter to the project team (fulfilled here by the project manager) to handle a particular bug report. The first step in the process is for the submitter to input the details about the bug. This information is kept in the colloquy's shared data area.

After the submitter submits the report, the Test Lead is responsible to "reproduce" the bug, that is, to verify that the anomalous behavior can be reproduced from the instructions in the report. Failing to reproduce the bug usually means that the report is missing some detail and is therefor sent back to the submitter to supply the missing information and to submit it again. Notice that the use of loops like this make the plans non-deterministic and a better fit to the real world. The capability to construct such quality loops answers many of the problems discovered with linear oriented work flow systems.[16]

The request to reproduce the problem comes from the Project Manager to the Test Lead. Notice that this happens without the Project Manager needing to be aware that the bug has been submitted. This is because the submitter has followed the policy set up by the Project Manager. This is how the model assures that proper authority is presented when the proper process is used.

No subplan - Manual Operation

When the "Can Problem be Reproduced?" stage is activated without a subplan, the user must respond manually. The colloquy appears on the Test Lead's "pending" list, indicating that the Test Lead has responsibility for some action within the colloquy.

In the colloquy window, the Test Lead must respond to the request from the Project Manager by selecting either a menu item that says "Can not reproduce" or "Reproduced." These menu items were generated automatically from the VPL description of the plan. "Accept" & "Decline" built-in actions are also available in any stage so that the Test Lead can communicate whether the task will be done or not. Note that declining at this stage will bring the Project Manager into the colloquy for the first time, to resolve the problem. Eventually the Test Lead determines whether the problem is reproducible or not, selects the appropriate menu option, and the plan takes its course.

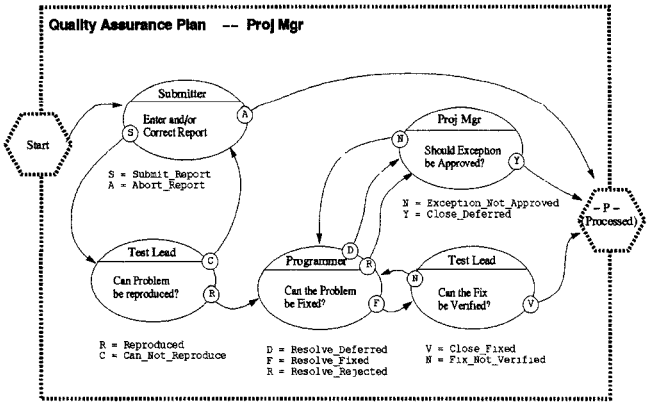

Figure 2. An example plan: quality assurance

Automation through Subplans

Responding to this request may take a complex process, so the Test Lead may wish to create a subplan. When starting to create a subplan for "Can Problem Be Reproduced?" a screen similar to that shown in Figure 3 will appear. The possible options that the parent stage expects appear in the exit nodes on the right. The plan is built by adding in stages, and assigning them to roles. The plan need not be complete before enactment can start, because editing on the fly allows it to be completed after starting. Ultimately, activating an exit node will cause the corresponding option in the parent stage to be taken.

Automatic Subplans from Policies

The Test Lead may find that he often receives the request: "Can Problem Be Reproduced?" Manually creating the plan every time could become quite tedious. Instead he will save one of his plans as a policy. The policy will create a complete plan automatically when it is invoked. The policy is just a template for a plan; it is the starting configuration for the plan, which may then be freely modified to fit the needs of the instance.

The policy can be set to be invoked automatically when the request is offered, or to wait until the assignee has accepted the task before creating the subplan. The latter option is so that the person might have a chance to review the details before accepting what might be a very complex task. If automatic invocation is chosen, then the assignment of subtasks could be passed on to others without the Test Lead being involved at all.

Incremental Introduction - Key to Success

Allowing incremental automation is a key to acceptability of the model and the system. Without automation the system works like e-mail: a request is sent from one person to another while retaining the benefit of the shared space and the history mechanism. Then, as users become more sophisticated, or tasks more repetitive, users may automate their own tasks, by creating plans and finally policies using VPL. Tasks that are rare and are not worth the trouble to automate do not need to be automated.

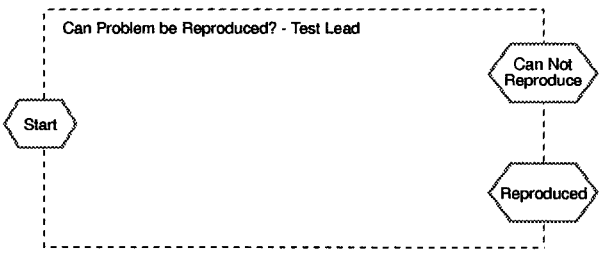

Figure 3. An empty subplan diagram

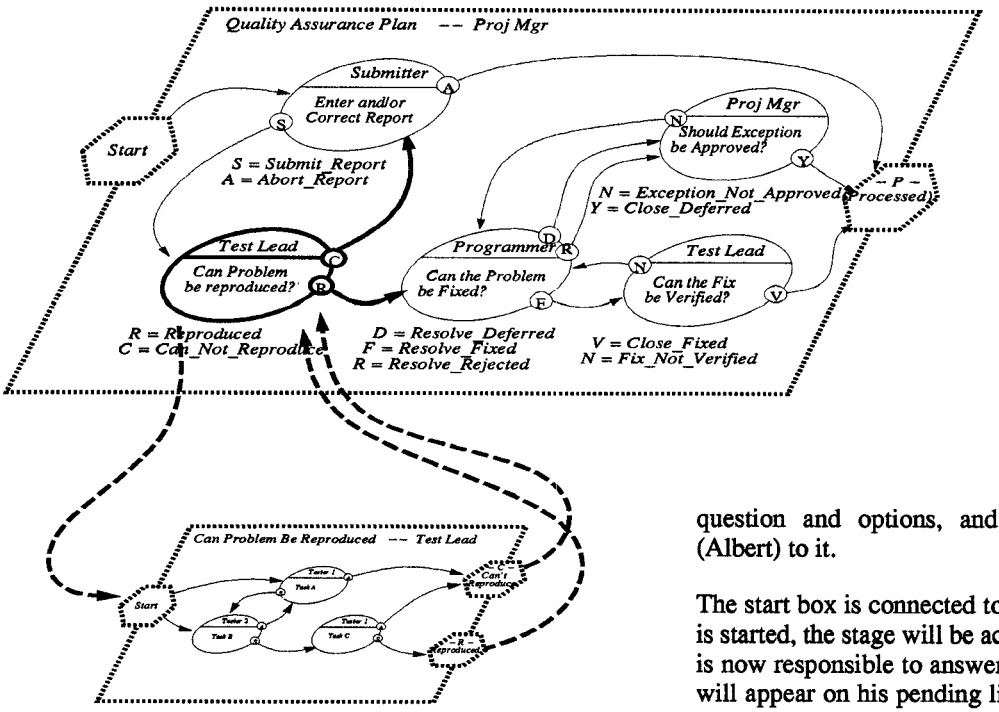

Diagram 1. The upper plan, owned by the Project Manager, represents his view of the process, and only he may modify that plan. The highlighted active stage represents a request from the Project Manager to the Test Lead. The subplan has been created by the Test Lead in order to satisfy this request. The Test Lead holds exclusive right to modify the subplan. Results in the subplan effect transitions in its parent.

It is also critical that the person who does the automating is the same person who sees the benefit from this. You automate your own tasks. the more time you put into it, the more benefit you receive. Lack of observation of this principle is cited as a reason for failure of many groupware systems.[6][7] A similar plea for empowering user is made in [1]. VPL empowers users to have control over their individual parts of the process, allowing them to experiment with, modify, and evaluate the processes, leading to real business process reengineering.

PATTERNS OF USAGE OF SUBPLANS

In order to illustrate the ad-hoc flexibility and expressiveness of VPL, a number of commonly occurring cases have been included in this section. All cases are examples of how a Test Lead named John might automate the answer to "Can the Problem be Reproduced?" These are examples of policies that he might write to be automatically invoked.

Delegation: Figure 4 shows a simple, and probably common, usage of a subplan to delegate the task by requesting that someone else handle it. This is the way that requests would be automatically routed to a person who specializes in that task. This is done by drawing in a stage with the same question and options, and assigning a different person (Albert) to it.

The start box is connected to the stage so that when the plan is started, the stage will be activated. The effect is that Albert is now responsible to answer the question, and this colloquy will appear on his pending list. When he does answer it, his answer causes the conclusion of the subplan by activating the appropriate exit node.

Rephrase the Question: The request "Can Problem Be Reproduced?" comes from the Project Manager's point of view of the project. It is often the case that within a different team, different terminology may be used.[5] John may decide to handle the request by requesting a different task, one more tuned to the way the test team works. For sake of example shown in Figure 5, this team considers a problem report to be *valid* if it is reproducible, and *invalid* if not. The question that Albert will see is "Is this problem report Valid?" The answer is a simple "Yes" or "No." The yes or no response is translated to the "Can Reproduce" or "Can not Reproduce" response by connecting them with an action line. The project manager is not required to learn the special terminology of the team. This rephrasing ability allows for a matching of the way that different groups work and view the task.

Serial Decomposition of Task: What may be a single step at one level, may be several steps at another. The example in

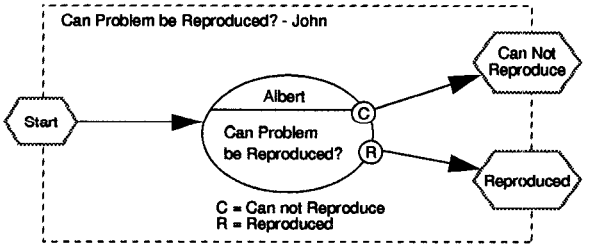

Figure 4. Simple delegation of identical task

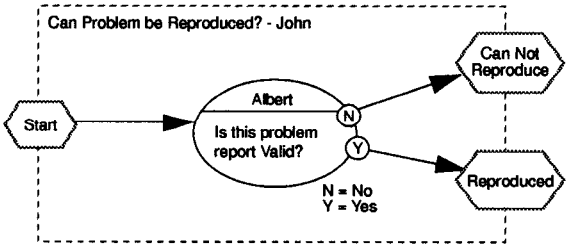

Figure 5. Delegation with rephrasing the question

Figure 6 show linear decomposition of the task into three distinct steps. These steps could be assigned to different people, or to a single person as a simple sequence reminder. As each stage completes, it activates the next stage. When the final stage completes the entire plan is completed, and the result is reflected to the higher level. Each stage also involves a Yes/No choice; at any point if the "No" option is chosen, the subplan is exited without activating remaining stages.

Parallel Decomposition of Request: The example shown in Figure 7 shows two separate tasks started in parallel. Two different testers see this colloquy appear on their pending lists simultaneously. They are both requested to perform the test on two different systems, and they both make a decision. If either of the testers can positively reproduce the bug, this conclusion immediately causes the completion of the plan. This OR condition is denoted by having the action arrows point directly to the exit node, which terminates all stages within the plan if there are any left active. The AND-node is used to prevent the subplan from being terminated when only one tester can not reproduce the problem. They must both conclude that the problem can not be reproduced before the plan is terminated sending a "Can Not Reproduce" event to the parent.

Parallel decomposition allows for several tasks to be started without any sequencing constraints upon them. The people (or teams) are able to work concurrently.

Conditional Branching: Figure 8 shows an example where the plan tests the value of the "system" attribute entered by the submitter. If system is set to Unix, then Albert is requested to validate the report. If system is set to MSW the Joe is requested to validate the report. Only one of them will see the colloquy appear in their pending list, and the answer that they give will become the result of the subplan.

If neither of the conditions is true, then the plan is set to automatically return that the problem can not be reproduced, presumably because these are the only two systems that John can test. "Can Not Reproduce" routes the responsibility back to the submitter, who may change or correct the value for "system" and resubmit the problem report.

Delegation with review: Many people will feel uncomfortable with blind delegation, especially when the result is to go back to their superiors. Figure 9 shows an example where John requests Albert to handle the request, and his answer is reviewed by the John before the final conclusion of the plan because the Alberts actions trigger stages (that are, in effect, a request from John to himself) that inform John of the result. He can choose "OK" to make the answer official, or ask Albert to reexamine the problem report again. Keep in mind that with each action, a message can be added to explain the reasons for the action, in this case why John feels that the problem should be looked at in more detail, or perhaps just to ask Albert to include more evidence to support his conclusion. Albert and John can continue in the loop until they are both satisfied with the result.

Delegation to anyone in a group: If the stage is assigned to a group, then anyone from that group may accept the task.

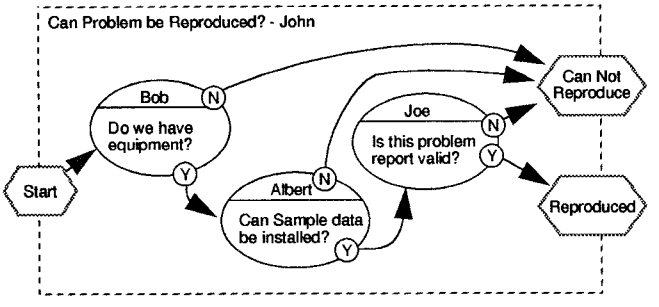

Figure 6. Decomposition of task into 3 serial tasks.

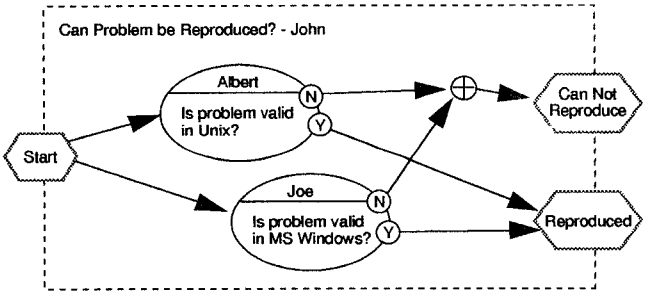

Figure 7. Two tasks activated in parallel

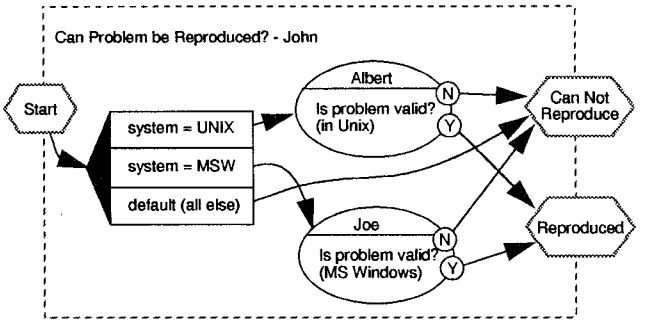

Figure 8. Conditional branching before delegation.

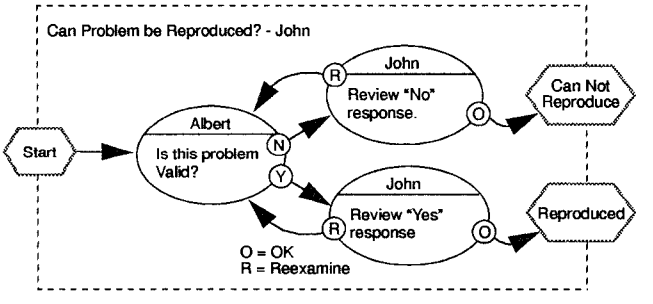

Figure 9. Delegation with review

Figure 10 shows an example of this. While the stage is in the "offered" state, until it is accepted by someone, the colloquy will appear on every member's pending list. Anyone in that group may take accept the task, acting as that group's representative. A good example of where this behavior is needed is in a customer support group where customer requests come in asynchronously, and the first available person picks up the task. As soon as one person accepts the request, the stage becomes the responsibility of that person, and the colloquy will disappear from the pending lists of the rest of the group. Alternately, if the request is a sufficiently simple one, any member of the group may immediately take the action, and the result will send an event to the exit node, causing the plan to be terminated,.

Delegate to everyone in a group: Figure 11 shows an example where a request is made to each and every member of a group through the use of a parallel split stage. Internally, the system will make a copy of that stage for every person in group, and then start them all in parallel. The behavior here is that when one person takes an action, it does not affect the copies of the stage for the others; they may still take their actions independently. But as soon as one person answers Yes, the entire plan is completed, and the remaining stages are terminated thereby removing their option to respond.

This plan is just like that of Figure 7 except that the number of parallel tasks is not determined until the stage is activated. Similarly, the AND node has been used to avoid termination on "No" answers unless all people answer "No". On the other hand, as soon as one person answers "Yes" the plan is completed and the result communicated to the parent stage. Other behaviors can be programmed if desired.

Delegation with reminders: Figure 12 shows an application of the timer-node; a request is delegated to Albert, but John has set it up so that if the task has NOT been completed after

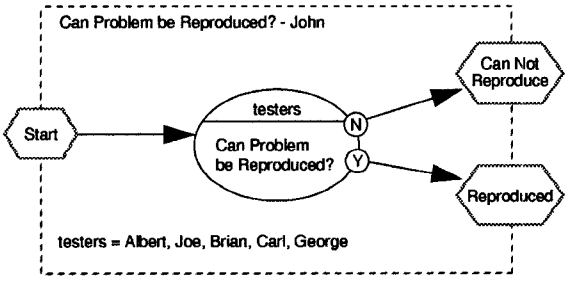

Figure 10. Assignment to anyone in a group of people

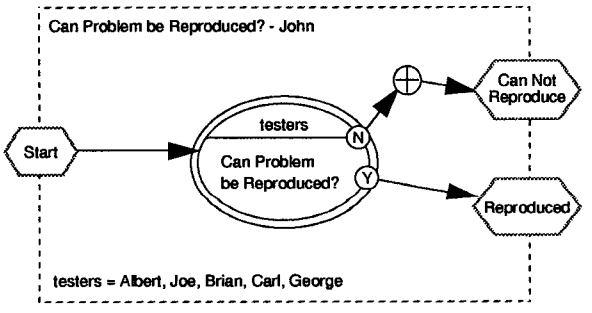

Figure 11. Assignment to every individual in a group

1 week, a reminder to "Check into Progress" stage is activated. This stage will be activated in parallel with the one assigned to Albert, but since it is assigned to John, it will make the colloquy appear on John's pending list, making him aware of the colloquy so he can call it up to be displayed on his screen. Remember Albert may have a subplan that delegated the task to someone else. John can browse the records of the activity so far. By being able to see what has been happening, much of the "Why isn't this done" communication is avoided. John has given himself two choices. He can "Reset" the clock and wait another week, or possibly "Abort" and give up the search for the problem.

DYNAMIC MODIFICATIONS AND SYNCHRONIZATION

The previous section gave some examples of the expressiveness of VPL in typical ad-hoc situations, yet each of the examples has plans that were essentially static. This section gives examples of how a user might modify the plan while it is in process in order to handle exceptional situations. The previous examples showed how the assignee, or recipient of the request is able by creating a subplan to add stages to or change the flow of the process without modifying the parent plan. This section deals with how the requester, or owner of the plan, might modify plan to make it fit better to the instance.

The VPL / script combination can be considered to be reflective in that scripting commands embedded within the VPL structure can create new stages, delete stages, add or delete actions from existing stages, and programmatically modify the plan in many ways. Plans that modify themselves are difficult to visualize and understand. This is contrary to our goals of making processes visible and easy to understand. If the exception is known ahead of time, it is wiser to incorporate the handling into the plan in a more visible fashion. Nevertheless, it is worth noting that the all of the actions discussed below could be incorporated into scripts that might be operated by the user as macros to simplify the task of responding to unforeseen situations.

Exception Handling: Regardless of how good one tries to make a policy, there will always be real life cases that do not fit.[27] The process might be progressing fine until the user unexpectedly finds that before the currently active stage can be performed, some extra task must first be completed. If that user owns the policy, or if the user involves the owner by declining the task, a new stage can be added to the executing plan dynamically. In Figure 13 Jeanette wishes to

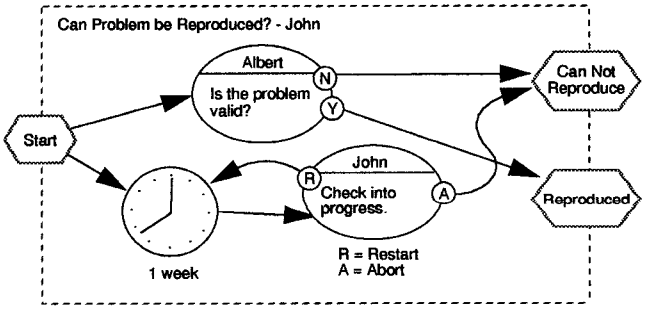

Figure 12. Delegation with delayed review

see if the same problem exists in, say, X11R5. As an alternative to sending this request by e-mail, she creates a stage asking for the work to be done. This not only communicates the request, but also gives Brian access to all the information about the problem report and the process up to that point. In doing this, the active stage is put into a state that waits for the result of the new stage and, in this state, no longer appears on Jeanette's pending list. Then, when the new stage is completed the original stage is reactivated and it reappears on her pending list so that she can see the response and go about handling the original task.

Wait for a stage to complete: Figure 14 shows an example where a new typesetting system has been ordered by the company. All documentation people have been asked to use this system for all future work. A document going through the process gets to the stage to be typeset. Since this stage can not be performed until the equipment is received, there is little point in having this task wait around in someone's pending list. It should disappear, and then reappear when the equipment arrives.

Instead of creating a new stage, this is done by placing an obligation on a stage in another colloquy (in this case the purchase order colloquy). The active "Typeset Document" stage is set to wait until it receives the event from the other stage. The obligation on the other stage will send the event when the stage it is placed on deactivates, which will happen when the equipment is received (or the purchase order process is cancelled) This has the effect removing the typesetting task from the pending lists, yet when the equipment arrives, it will reappear.

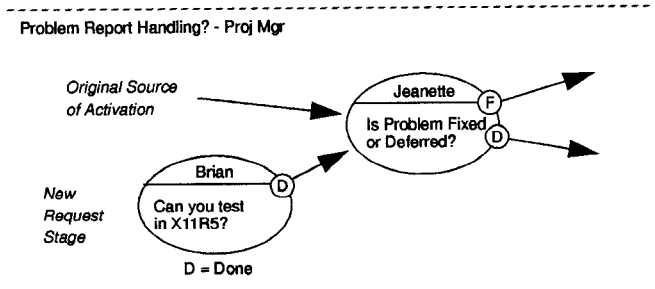

Figure 13. Addition of new stage to a plan in process

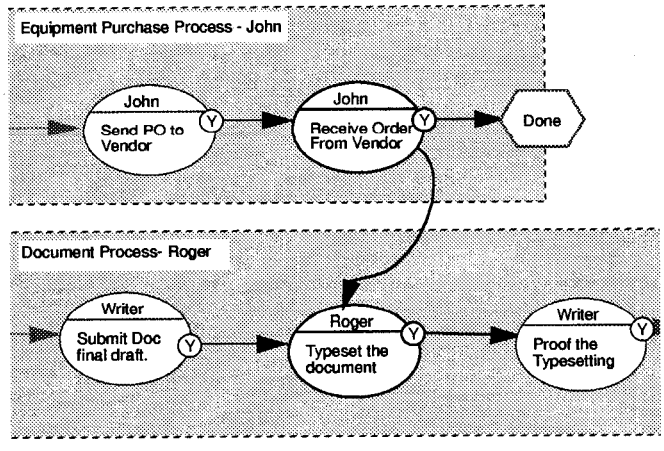

Figure 14. Wait for another stage to be completed

The real convenience of this form of coordination is that John does not need to *remember* to tell everyone that the equipment has arrived. He concentrates on his own job, and the system supports the communications automatically.

Wait for a stage to activate: In order to synchronize a stage to when another stage starts, an obligation may be placed in a manner similar to that described above, except that the obligation is triggered upon activation of the stage as opposed to the deactivation of the stage.

The only modification that an individual can make to a plan that they do not own is the placement of an obligation. Anyone may place an obligation on any stage in the system to send an event when that stage activates or deactivates. Because of this Roger was able to place the obligation without disturbing John. This mechanism allows for a very flexible synchronization of processes within the system.

Send the result of the stage: Instead of simply receiving an activate event, the obligation can be placed such that it communicates to the waiting stage whatever event deactivated the other stage. A good example here is when two bugs that appear different externally are found to be duplicates of each other. Neither report should be discarded because when the problem is fixed, the testers should verify that the fix worked in both cases. But the programmer may decide to defer the fix, in which case both reports should be deferred. By linking the two processes together with an obligation that communicates the result of one stage to the other as shown in Figure 15, both bug reports are guaranteed to be handled the same way. If the first bug is fixed, then the second bug is fixed. If the first is deferred, then the second is deferred. The actions of two different colloquies are synchronized.

Combinations: Combinations of the above will form a flexible way to coordinate activities. For example, the user may wish to make one stage wait on the result of another stage, although the actions of the stages are not congruent. By creating a new stage in the colloquy that accepts the same

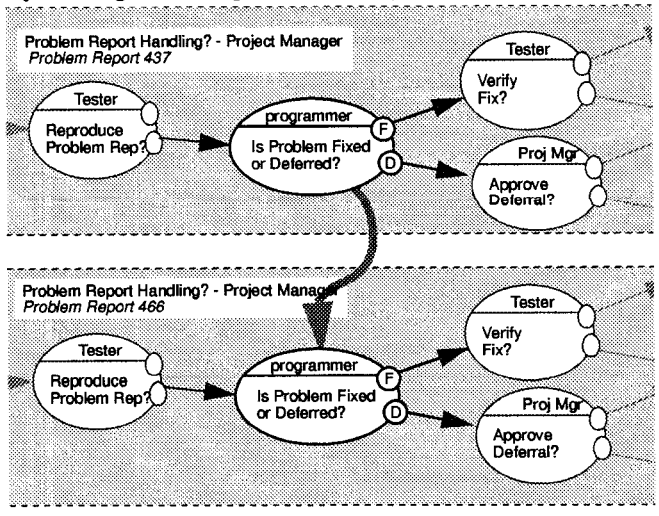

Figure 15. Wait for result of another stage

event, the events can be translated into the expected events. In Figure 16, the Leader of a group has reached a point where a decision needs to be made about whether to call a meeting. He realizes that Alex is considering going to a conference, but Alex has not yet made up his mind. Furthermore, there is no point in having the meeting if Alex will not attend. Since he has already decided that the meeting should be called if and only if Alex does not go to the conference, he wishes to make this happen automatically. He creates a translation stage that will receive whatever event deactivates the conference question stage. Then, on receipt of a Yes event, a No is sent to the original stage, and vice versa. The meeting will be automatically called if Alex decides not to go.

COMPARISON TO OTHER SYSTEMS

Workflow packages typically model work as documents which travel from person to person, each person changing, consuming, or producing such documents and sending them on to others. One of the key benefits that information technology gives us is fast concurrent access to information regardless of location; modeling a system as moving data from one point to another works contrary to this benefit. The Regatta approach is to allow data to be ubiquitously available to all members of a colloquy. The completion of a task is not accompanied by a loss of access to the data manipulated by that task. Thus the flow of data is far less important, allowing Regatta VPL to concentrate on coordination of behavior.

Systems which model work using Data Flow diagrams are at a disadvantage because Data Flow diagrams give insufficient clues as to the sequencing of the events. Furthermore, it is difficult with data flow to describe different tasks being performed on the same artifact in parallel. Rather than data flowing from place to place, the analogy for Regatta is that the colloquy provides a shelf where all the documents are placed, and are concurrently available to all participants.

A process modelling system needs a way to break large steps into smaller steps when more detail is needed. Work breakdown structures and outline representations are common. The difference that Regatta brings is how tasks are broken into subtasks. If the process tool has a task orientation, then divisions of tasks into subtasks are made along functional lines. This is a problem because there are many different ways to decompose a task, and it is problematic to come to agreement within a group. Just as business processes are unknown, so are the individual discrete elements of a task.[31] In order to implement a task-oriented decomposition an analysis of the entire processes needs to be done before automating it. Decompositions appropriate for a large group, may be completely inappropriate for a small group or a single person. Instead, Regatta breaks stages into substages along the lines of speech acts, that is, requests from one person to another. While subtasks may or may not be observable, speech acts are always external observable behaviors. Though different workers within an office may not be able to agree on the steps to a task, they can however still agree on the meaning of a request. Beyond providing a way to decompose that a group can agree upon, it also divides the process into separate pieces which different people are responsible for. Regatta has made ample use of this advantage by allowing each user to have individual versions of a layer of decomposition.

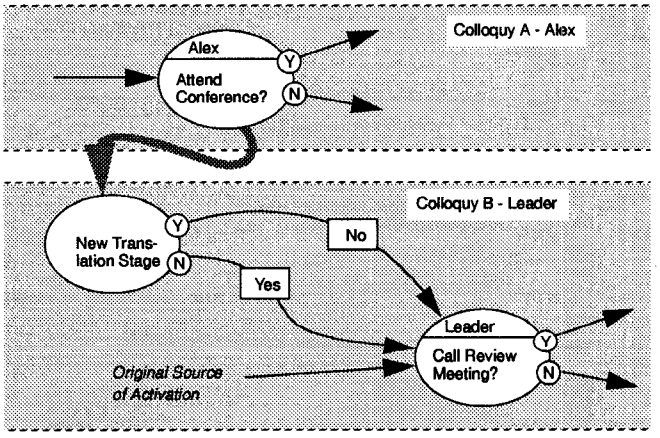

Figure 16. Wait for result of stage and then translate it

There are similarities to the Action Workflow™[17] by Action Technologies, especially with respect to the offer, accept, completion cycle. There are some distinct differences that should be pointed out. VPL diagrams allow the user to specify several different results, which result in different process paths. Action Workflow only allows graphically a single result: Done. Branches must be implemented in a non-graphical way. The satisfaction phase appears to be superfluous because from the diagram it is not clear what happens if the requester is not satisfied. Action Workflow places both ends of the request/response cycle in the same process model, while Regatta uses the request/response cycle to bridge different levels of responsibility, and therefor different plans. Request/response becomes what ties the parts of the process together, instead of the central elements of a single level of process itself. While Action Workflow diagrams are useful as descriptions of the process, they do not show the complete process, since branches, multiple responses, and exceptions must be implemented in a hidden way.

The pending list can also be compared to the in-box of a mail system for those companies that use e-mail to coordinate tasks. The e-mail based approach has two disadvantages. First, the e-mail in-box is private and there is no way for the others in the collaborative process to know whether a request has been handled, whether a user has passed the request on to others, or whether the task has started a series of subtasks. One of our goals is to let others know this information which is useful for coordination. The other significant disadvantage is that once a message is placed in an in-box it can not be removed except by that user. This means that you can not ask a question to a group of people then retract it when it is answered. Regatta allows tasks to be cancelled or reassigned, thereby moving it from one person's pending list to another.

A lot of interesting work has been done in the area of software process definition and improvement. Much has

been done with state charts. State charts provide a powerful way to model complex mechanical systems, but they were not designed to model human interaction. (For example, turn on a light switch and the light may fail to turn on, but it never *declines* to turn on.) Speech act orientation makes Regatta technology more suited for collaborative work.

CONCLUSION & PROJECT STATUS

A system that implements this model is in beta test. The server and client both run on a Unix workstation (SPARCstation for now) with an XWindows user interface. An MS Windows supported client is expected in 1994.

Experiments with the existing system have shown that the visual formalism is sufficiently powerful to model a wide variety of business processes. Inexperienced users are able to read the diagrams after only a few minutes of explanation about how they work.[28][29]

When building new policies, users were generally able to create single level policies without trouble. Users typically implemented "flat" policies that included all of the details of the process at a single level, which because of the demands of a real world process got fairly large. Yet after some experience, users were able to allow parts of the plan to be defined and controlled by other parts of the organization in sub plans. The recognition of the fact that part of the plan was the responsibility of another group was not initially obvious; users started out to be what can best be described as rather "control-oriented". It is important to keep in mind that the computer can never force someone to do something -- only the social conventions influence a person's behavior. To be useful, the system must be a credible conveyance of such conventions.

The Regatta Vision

In summary, the model and visual language are appropriate to accomplishing the three goals:

1. Coordination: business processes may be automated; users can find tasks that are waiting for them to do; exceptions can be handled in an efficient manner.

2. Understanding Processes: discovery of the process is supported by allowing activation of incomplete plans that may be later modified on the fly; a history mechanism records what actually happened; the plan in its current state is depicted in a graphical form, with the roles, tasks, and responsibility made clear; built in help system explains new processes or changes in a process to users.

3. Change: VPL enables a greater number of users to program processes; processes are partitioned into plans at different levels to allows each user to plan their part of the process from their viewpoint, and to allows different people to control different parts of the process.

Goals two and three are complementary; while an understanding of the process helps point out areas of potential improvement, it is imperative that after a modification the system help explain the change, so that people can work effectively within the new process. A continual cycle of experimenting with new processes and observing the results will lead to what has been called a "Learning Organization"[24]. This is a new breed of organization transformed by information technology to be truly dynamic, highly efficient, and able to respond quickly to today's increasingly unpredictable external pressures.

ACKNOWLEDGEMENTS

The author would like to thank Robin Maxwell, Toshikazu Matsumoto, and Bahram Saghari for help in developing and refining these ideas, and for developing the system which implements this model; Bill Talone for his critique of an early draft of this paper; Simon Kaplan and the ConversationBuilder team for discussions about the usability of the model, and for implementing an early prototype in ConversationBuilder; Robert Hotchkiss & Jim Larson for proofreading the final draft and providing suggestions to make things clearer, and to many people who actually tried to use the system and responded with helpful comments.

REFERENCES

1. Andrew Clement, Computer Support for Computer Work: A Social Perspective on the Empowering of End Users, *CSCW 90 Proceedings*, ACM Baltimore MD, 1990
2. Thomas H Davenport, James E Short, The New Industrial Engineering: Information Technology and Business Process Redesign, *Sloan Management Review*, Summer 1990.
3. Gartner Group, Inc. Business Process Re-engineering *SPA-210-590*, August 7, 1991
4. Saul Greenburg, *Computer Supported Cooperative Work and Groupware*, Harcourt Brace Jovanovitch, Academic Press, 1991
5. Ray Grenier and George Metes, *Enterprise Networking: Working Together Apart* Digital Press, 1992.
6. Jonathan Grudin, Obstacles to user involvement in software product development, with implications for CSCW, reprinted in [4]
7. Jonathan Grudin, Why CSCW Systems Fail, *Proceedings of the 1988 Conference on Computer Supported Cooperative Work*, ACM, p85-93, Portland Oregon, 1988
8. Keith Hales, Mandy Lavery, *Workflow Management Software: The Business Opportunity*, Ovum Ltd. December 1991
9. Michael Hammer, Re-engineering Work: Don't Automate, Obliterate, *Harvard Business Review*, July/ August 1990
10. David Harel, On Visual Formalisms, *Communications of the ACM*, 31(5):514-530, May 1988
11. Carl Hewitt, Offices are Open Systems, *ACM Transactions on Office Information Systems*, 4(3):271-287, July 1986

12. Robert J. K. Jacob, A State Transition Diagram Language for Visual Programming, *IEEE Computer,* 18(8):51-59, August 1985

13. Simon M Kaplan, William J. Tolone, Douglas Bogia, and Celsina Bignoli, "Flexible, active support for collaborative work with Conversation Builder", *Proceedings of the 1992 Conference on Computer Supported Cooperative Work*, ACM, 1992

14. Simon M Kaplan, Alan M Carroll, Kenneth J MacGregor, Supporting Collaborative Processes with ConversationBuilder, *Proceedings ACM Conference on Organizational Computing Systems*, p69-79, November 1991

15. Simon M Kaplan, Keith D Swenson, Operating System Support for Collaborative Work, *Proceedings for the Second International Workshop on Object Orientation in Operating Systems*, September, 1992

16. Thomas Kreifelts, Elke Hinrichs, and Karl-Heinz Klein, Experiences with the Domino Office Procedure System, *Proceedings of the Second European Conference on Computer Supported Cooperative Work (ECSCW '91)*, p117-130, Amsterdam, September 1991

17. Raul Medina-Mora, Terry Winograd, Rodrigo Flores, Fernando Flores, The Action Workflow Approach to Workflow management Technology, *Proceeding of the 1992 Conference on Computer Supported Cooperative Work*, ACM, 1992

18. Michael S Scott Morton, *The Corporation of the 1990s, Information Technology and Organizational Transformation*, Oxford University Press, New York, 1991

19. John K. Ousterhout, Tcl: an Embeddable Command Language, Computer Science Department, UC Berkeley. Information on this can be retrieved from the Sprite Project, at sprite.berkeley.edu.

20. Sunil K Sarin, Kenneth R Abbott, Dennis R McCarthy, A Process Model for Supporting Collaborative Work, *Proceedings ACM Conference on Organizational Computing Systems*, November 1991

21. Michael Schrage, *Shared Minds: The New Technologies of Collaboration*, Random House, New York, 1990

22. John R Searle, "A classification of Illocutionary acts", Language in Society, 5 p1-23, 1975

23. John R Searle, *Expressions and Meaning: Studies in the Theory of Speech Acts*, Cambridge University, Cambridge, 1979

24. Peter M. Senge, *The Fifth Discipline: The Art and Practice of the Learning Organization* Doubleday/ Currency, New York, 1990

25. Nan C. Shu, FORMAL: A Forms-Oriented Visual Directed Application Development System, *IEEE Computer*, 18 (8): 38-49, August 1985

26. Paul Strassman, *Information Payoff: The transformation of work in the electronic age*. Free Press, New York, 1985

27. Lucy Suchman, The role of common sense in UI design, *Highlights on the International Conference on Office Work & New Technology*, Cleveland, 1983

28. Keith D Swenson, The Regatta Project, *Proceedings of the First International Conference in Technologies and Theories for Human Cooperation, Collaboration, and Coordination*, Applica '93, March 1993

29. Keith D Swenson, A Visual Language to Describe Collaborative Work, *Proceedings of the International Workshop for Visual Languages*, Bergen. Norway, Aug 1993.

30. Workflow, Groupware, and Re-engineering:, *IT Horizons*, 1(3):1-11, September 7, 1992

31. Shoshana Zuboff, *In the Age of the Smart Machine*, Basic Books, New York, 1988.