Copyright 1998 IEEE. Published in IEEE Internet Computing, Jul/Aug 1998. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 732-562-3966.

An architecture and environment for decentralised, internet-wide software process modelling and enactment

John C. Grundy[†], John G. Hosking[††], Warwick B. Mugridge[††] and Mark D. Apperley[†]

[†]Department of Computer Science, University of Waikato
Private Bag 3105, Hamilton, New Zealand
{jgrundy, mapperle}@cs.waikato.ac.nz

[††]Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
{john, rick}@cs.auckland.ac.nz

Abstract

Centralised client/server approaches to supporting software process modelling and enactment are common, but can suffer from serious robustness, performance and security problems. We describe a decentralised architecture for software process modelling and enactment, which also incorporates distributed work coordination, task automation and system integration facilities. Our environment based on this architecture uses visual, multiple view specifications of work processes, together with animation of process model views to support enactment awareness. This environment provides a robust, fast and secure system for coordinating work on distributed software development projects utilising basic internet communication facilities.

Keywords: process modelling and enactment, decentralised software architectures, computer-supported cooperative work, work coordination, distributed software agents

1. Introduction

Process modelling and enactment tools support cooperating workers in defining software process models which describe the way they work, or at least how they should work, on distributed software development projects [2, 15]. Running, or "enacting", these models allows developers, whose work may be distributed over time and place, to more easily follow prescribed or recommended processes and to more effectively coordinate their work [5, 8]. These process support tools may also include software "agents", used to automate simple tasks, support work coordination, integrate use of disparate tools, and track and record histories of software development work for future reference and analysis. Many development teams, especially distributed teams, require such support to adequately coordinate their complex, distributed work practices.

Most existing process support tools, and most cooperative work supporting tools in general, use centralised, client-server software architectures [2, 15], and often can only be used with dedicated local area networks. A central server permits collaborative process modelling, and a centralised process enactment engine runs ("enacts") these process models, records work, and supplies task automation and systems integration agents. However, research has shown that such centralised architectures can have serious robustness, performance, security and distribution problems [1, 6]. Attempts have been made to provide distributed process enactment engines, to overcome some of these problems, and some work has been done in facilitating distributed task automation [1, 6]. However, little work has been done to facilitate distributed work process modelling and more generalised software agents.

We have developed a distributed architecture and environment for fully decentralised software process modelling, process enactment and distributed work coordination, task automation and tool integration agents. Our architecture provides significantly improved robustness, performance, security and distribution capabilities over centralised process support tools. It can be run over local area networks, internet wide area networks, and modem and mobile computer networks, and supports a wider range of distributed collaborative process modelling and software agents than most other distributed process support systems.

2. The Serendipity-II Process Management Environment

We have been developing process modelling and enactment environments for defining work processes for a variety of professional domains, including software engineering, patent law, real estate, and immigration consultancy, and a variety of office automation tasks, including inventory management and project management [8]. We developed the decentralised Serendipity-II process modelling environment for use on these tasks, an improved version of an earlier, central-server based process support tool, Serendipity [8]. Serendipity-II provides multiple views of process models using a range of mainly visual languages. Figure 1 illustrates the basic process modelling and enactment capabilities of Serendipity-II. The left view shows a simple software process model for modifying a software system. Ovals represent process stages, and include a unique name and role performing the work. Enactment flows, which may be labelled, connect stages and pass on "enactment events", driving process model execution.

Stages may have subprocess models defined, such as the bottom right view of Figure 1, which shows the subprocess for the "*2. Design changes*" process stage. Input and output event icons, shown as hexagons, indicate where enactment flows enter and leave subprocesses, and constrain the process stage these enactment events flow into or out from. Role assignment views, such as the one in the top right view, allow process modellers to associate particular users, or software agents, with stage roles. Other views supported include usage of tools and artefacts by process model stages and complex artefact structure definitions. The dialogue at centre top shows the enacted stages for each user (here only "john", the project leader, is doing work for this process model - planning specific tasks for his co-workers).

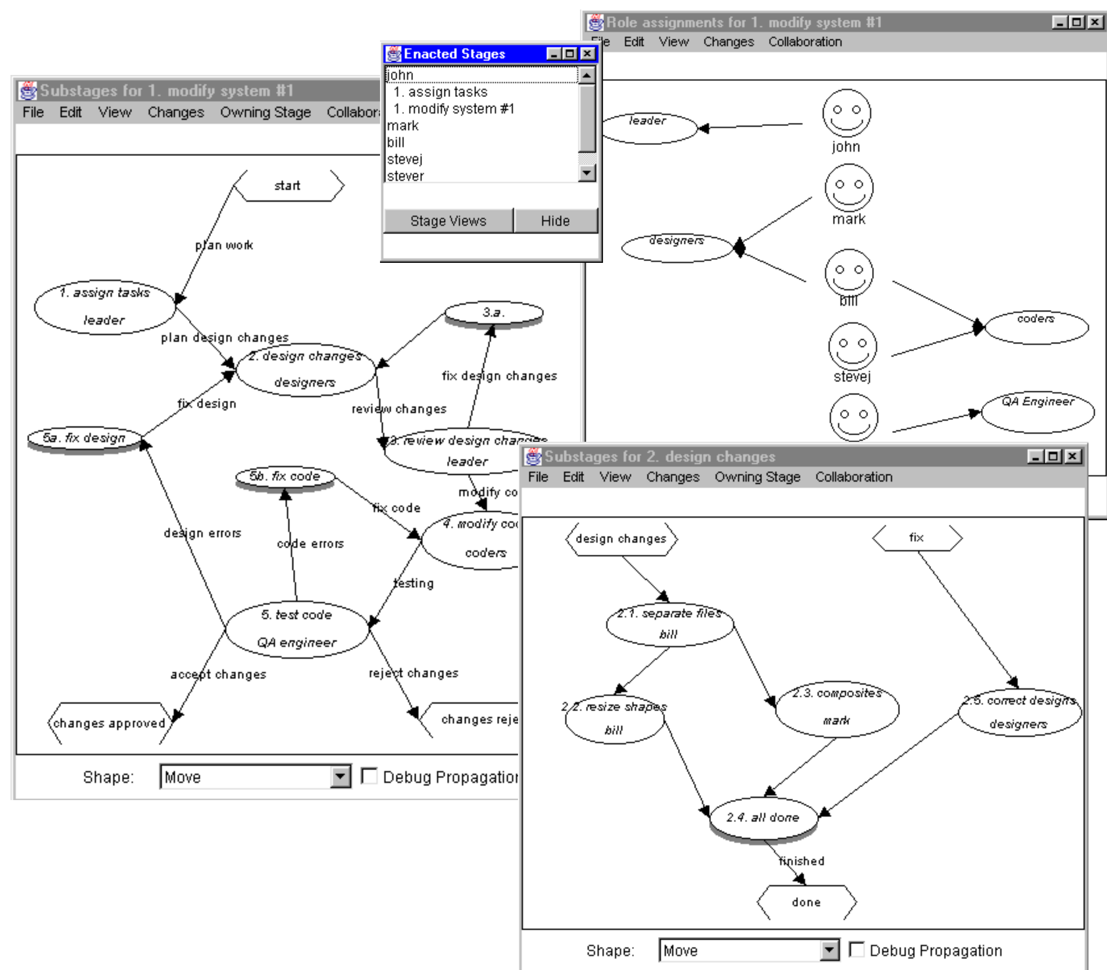


Figure 1. An example of process modelling and enactment in Serendipity-II.

In order to provide robust, efficient, secure and distributed process support using Serendipity-II's capabilities, we required an architecture for this environment that would satisfy the following requirements:

- Users are able to collaboratively edit process models both synchronously (i.e. what-you-see-is-what-I-see style) and asynchronously (i.e. editing alternate versions with subsequent version merging). These collaborative editing, and supporting communication capabilities (audio, text chat, messaged, annotation etc.) should be decentralised to ensure robustness and efficient performance.

- Users are able to enact process models in a decentralised way i.e. have their own process enactment engines. Various awareness capabilities should enable users to track each other’s work e.g. which process stages they are currently working on, which others are enacted, histories of enactments and work, etc.
- Decentralised work coordination agents should be supported. Local agents should not need to access other users' process models and enactment information, and agents coordinating multiple users should communicate in a decentralised way (running either on a particular user's machine or as independent "environments").
- Users must have full control over access by others to their process models, and the deployment of software agents which affect their work.
- Process modelling, enactment and work coordination agent facilities must work equally well over both high- and low-bandwidith network connections, and must be tolerant of periodic disconnection from the network.

3. A Decentralised Process Management Architecture

We have implemented the Serendipity-II process modelling and enactment using the JComposer metaCASE tool and the JViews object-oriented framework [9]. JViews is implemented in Java and provides a component-based software architecture for building multi-view, multi-user environments, extending the Java Beans componentware API. JComposer provides visual languages supporting the specification of tool repository and view components, and the specification of editor icon appearance and functionality. JComposer generates JViews class specialisations, with tool developers able to further extend these generated classes to code complex functionality in Java.

Figure 2 shows the basic process model information for Serendipity-II being specified in JComposer. JComposer provides basic abstractions of "components" (rectangular icons), relationships (ovals), attributes and links (labelled arcs). Base (i.e. repository) information for Serendipity-II includes stages and enactment links, together with the tool repository itself and a hashtable relationship linking all base stages to the repository. Multiple views of this repository can be specified, adding additional information about the process modelling language, as well as definitions of multiple representations (views) of the repository information [9]. JViews classes implementing Serendipity-II process modelling capabilities are generated from these JComposer specifications. We have extended these generated classes to incorporate process enactment capabilities by hand-coding using Java.

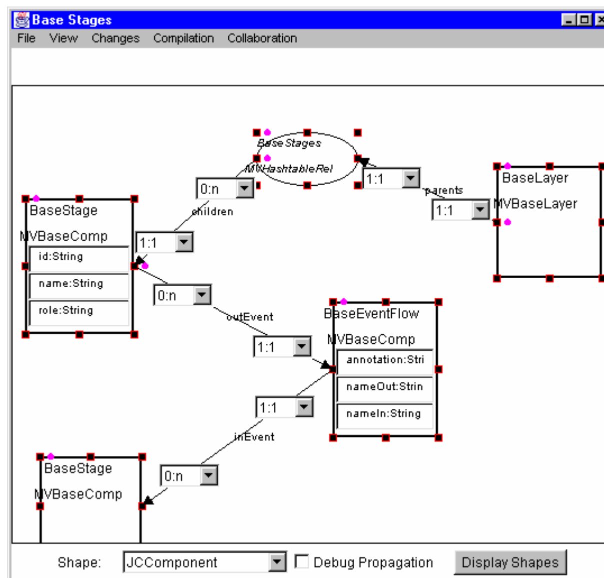


Figure 2. Simple Serendipity-II repository components being specified in JComposer.

The generated JViews implementation of Serendipity-II represents process stage and enactment information as components, liked by relationships. When a component is modified, a "change description" object is generated, documenting this state change event. Change descriptions are propagated to relationships which pass the change event onto other components, carry out processing in response to the event (e.g. enforcing constraints or updating multiple views of a component), or ignore the change [7]. We have used this same change propagation mechanism to implement the process enactment engine of Serendipity-II, by representing enactment events as change descriptions and propagating these to connected components representing other process stages and links. JViews change descriptions can also be stored in "version records" to facilitate work tracking, undo/redo, versioning of components via deltas, and be broadcast to other users' JViews-based environments to facilitate cooperative work.

We have used JViews' component and change description serialisation and inter-environment propagation mechanisms to develop a decentralised software architecture for Serendipity-II, eliminating centralised servers. Every user's environment is responsible for; its own communication with other environments; its own process model enactment; and storing its own process model and enactment components. It may also have its own "local" software agents. Figure 3 illustrates this architecture.

Each user of Serendipity-II has a modelling and enactment environment that provides multiple views of work processes with which users interact. Each environment has its own "receiver" (server) and "sender" (broadcasting client) components, used to communicate with other users' environments. Thus rather than a centralised server, communication is via one user to zero or more other users. In large systems, communication can be from user to "groups" of other users i.e. forwarding agents are used to propagate information to groups of other users and/or distributed software agents. "Users" in our architecture may also include Serendipity-II, or third-party, software agents and Information Systems interfaces. Our architecture treats these in the same manner as components that interface to people. Both sender and receiver run asynchronously with editing and software agent processes, ensuring fast response time for user editing and good agent processing performance over both high- or low-bandwidth networks.

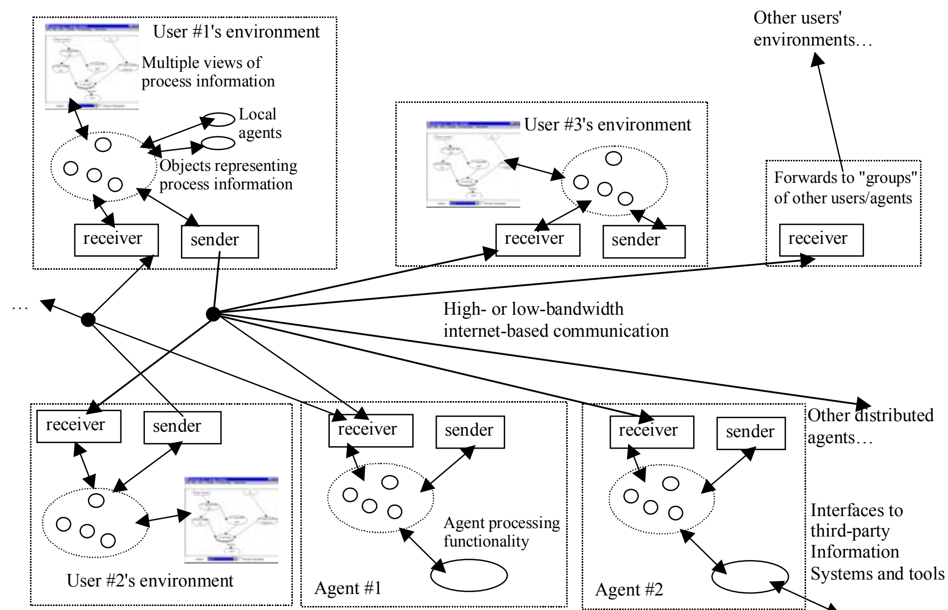


Figure 3. A decentralised process modelling and enactment architecture.

Component objects in our JViews architecture have system-allocated unique component identifiers. Each user is given an "identifier factory", ensuring all components created by a user's environment have a unique id for their own as well as other users' environments. Components also have an identifier indicating the component they have been copied from (if any), allowing simple mapping between components representing different versions of the same information to be performed. Environments exchange components, groups of components and change descriptions by using the JViews component and change description serialisation mechanisms. Serialised model and enactment information is sent to other users' environments by the user's sender component (currently using TCP-IP sockets). Receiver components either map the deserialised component and change description (event) component identifiers to components in their own environment, or create new copies for any components they do not have copies of.

Thus in our architecture, some components may be local to a user's environment, representing private process information. All shared process model information, however, is fully replicated between users' environments, using our JViews component identification and versioning scheme. We use this fully replicated architecture so that shared process information can be independently modified even if one or more users sharing the information are off-line, mobile or their network connections are temporarily lost. This also allows a seamless transition at any time between synchronous and asynchronous process model view editing.

It is possible to keep parts of a shared process model synchronised, if required. This is achieved by incrementally propagating change descriptions generated by components representing shared information between environments. When received, the component identifiers of these change descriptions are mapped to corresponding components in the receivers' environments, and these components are updated to conform to the changes made by other users. Alternatively, a modified set of components can be copied to other users' environments, replacing the old versions.

Sets of change descriptions representing changes made to parts of a shared process model by other users can also be incrementally merged with a user's existing model. These change descriptions are sent to the other user's environment, and the other user then requests the changes be actioned on the components in their environment representing their copy of the shared process model. We have a change object annotation facility which allows a receiving environment to detect the loss of change objects and to request they be resent if necessary.

Our component-based implementation has a further advantage in that interfacing to third-party tools, information systems, and software agents can be done using Java Beans' component interface mechanisms, or by providing reusable JViews components to interface to these systems. These interfaces can be used by software agents defined in Serendipity-II to facilitate heterogeneous tool integration.

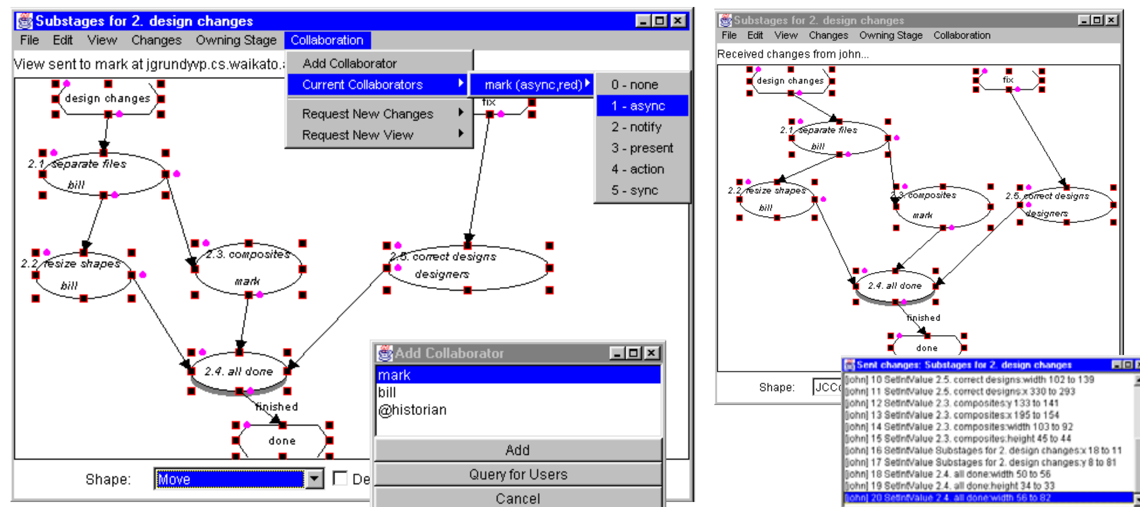
The following sections illustrate how we have used this decentralised architecture to provide a range of distributed process modelling and enactment facilities, and distributed work coordination, task automation and work tracking agents, over the internet in a decentralised fashion.

4. Collaborative Process Modelling

Serendipity-II supports decentralised process modelling. Users decide whether to collaboratively edit process specifications synchronously, or to independently model and evolve processes asynchronously, subsequently merging changes. Such temporary divergence is useful when evolving individual subprocess models. The ability to evolve process models when some users are off-line, using mobile computers or their internet connections temporarily lost is also increasingly important; the constraints of centralised process modelling interfere with these needs.

Collaborative process modelling and software agent specification are managed by broadcasting whole process model view definitions, or incremental changes to model views, between users' environments. A receiving user's environment maps changes to objects in the sending environment's view to objects in the receiving environment's view. The unique object copy identifier tags are used to map changes between alternate versions of the same object in the two environments. Asynchronous and synchronous editing differ in when changes are mapped.

Figure 4 shows an example of asynchronous process modelling. Users John and Mark have independently modified their versions of the process model shown. To reconcile their alternative versions, they can exchange whole copies of the modified view and manually reconcile them. Alternatively, they can exchange a list of change event objects (each describing an incremental change) which have been stored with the view. Each may then incrementally merge selected changes with their own version of the process model (as shown in the dialogue). This allows users to, for example, tailor aspects of a shared process model view (such as appearance and layout), or have different stages/enactment flows in their process model while it evolves. We have found this partial sharing approach to be useful during process model evolution. It is also useful if a particular user encounters exceptions in a process model which they wish to correct for themselves, but not have the effect immediately impact on others users' versions of the same subprocess. An additional advantage is one of control over one's own process definitions; users view and approve selected changes rather than have them always forced upon them by others' work.

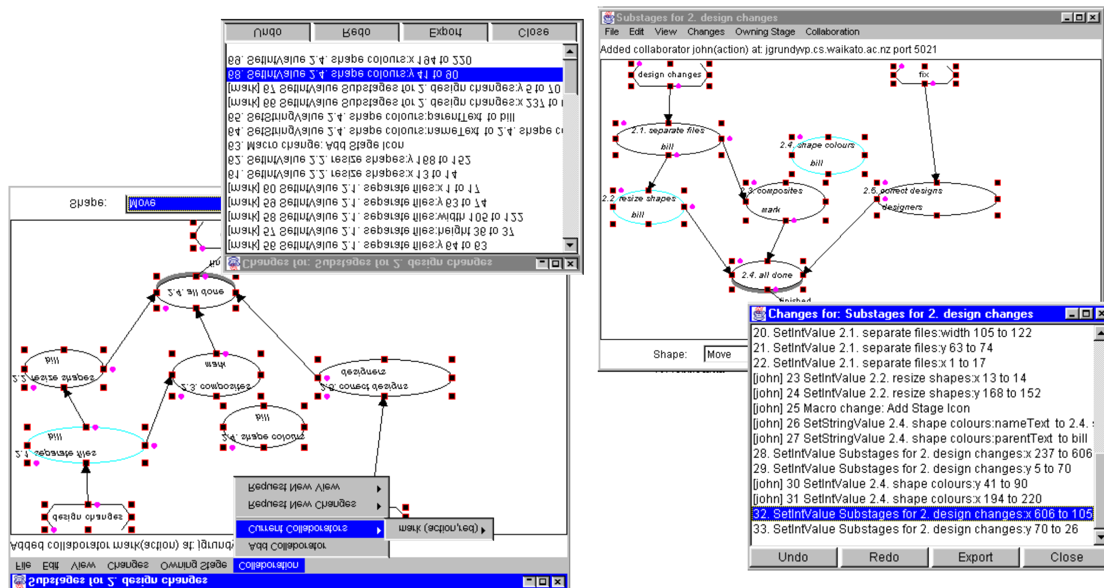


(a) John's view (b) Mark's View

Figure 4. Asynchronous process modelling example.

At times users desire closer collaboration over process modelling. In addition to asynchronous editing, Serendipity-II supports a range of "closer" editing modes ranging through to fully synchronous. *Present* mode allows users to be informed of changes being made to shared process model views by other collaborators as each change occurs. A description of each change is added to a dialogue, and users can choose to immediately merge the presented changes with their own process view, delay merging, or negotiate with other users about the change. In *action* mode all received changes are automatically applied (i.e. incrementally merged) as they occur. *Synchronous* mode similarly applies all changes immediately, but provides a locking mechanism ensuring there are no simultaneous updates by multiple users of the same view component. Locks are obtained by broadcasting a message to all users synchronously editing the view and obtaining a lock for the requesting user for the component to be edited from each environment. If two users simultaneously request a lock, it is denied to both and they must retry their edit, or coordinate their editing using chat, audio or other communication support.

Colouring of process model view icons is used to indicate the last user to change an icon in the view. Figure 5 shows users John and Mark synchronously editing a process model view. The changes in the view modification history are annotated with the name of the user who made each change.



(a) John's view (b) Mark's view

Figure 5. Synchronous process modelling example.

Our process modelling architecture is robust against users going off-line during collaborative editing. When they later return, their environment requests any changes made during their absence to be forwarded to them. They can then review and optionally merge these changes into their affected process model views. Our architecture also ensures ownership and visibility of views is in the control of the view creator, who decides who may be sent a copy of their views (using the "Add Collaborator" option in the Colloration menu, shown in Figure 5). Receiving users also decide whether or not they want their repository of process information updated to conform to a new view they have been sent. Fast semi-synchronous and synchronous collaborative editing performance is ensured as changes to view objects are only propagated to those other users (or distributed agents) interested in the changes. The asynchronous threading of the sender components for a user's environment broadcasts changes "in the background", ensuring view editing response for the user is maximised.

5. Distributed Process Enactment

A process enactment engine is included with each user's environment. Users may enact a process model stage (i.e. run it), finish a stage (say its complete), suspend or terminate a stage, or indicate the stage they are currently working on (which we call the "current enacted stage"). Each of these enactment activities generates an enactment events. These events may cause other stages to be enacted, finished etc. For example, finishing a stage in a particular "finishing state" enacts those connected stages that are specified to be enacted when the stage ends in this state.

Enactment events are propagated to other users' environments, informing them of the stage enactment and ensuring their process models are enacted appropriately i.e. flow-on effects of enactments are actioned in other users' environments. Enactment events are, however, only propagated to other users who have an "interest" in them, ie they have a copy of the subprocess enacted and have some role assignment in that subprocess. This reduces unnecessary enactment event propagation, and keeps enactment events for local subprocesses "private" i.e. prevents monitoring of users' "private" work.

Figure 6 shows a simple example of an enacted process model. John has finished planning tasks (stage "1. assign tasks"), and a finished enactment event has flowed into the "2. design changes" stage. This has resulted in the “2.1. separate files” stage being enacted for Bill. The stages in the process model views have been coloured to indicate enacted stages and the currently enacted stage for each user. An "enactment monitor" dialogue (bottom right) shows enacted stages i.e. assigned work for all users who have roles in the overall process model being used. The '*' indicates the currently enacted stage for a user. An enactment history dialogue (bottom left) shows the history of enactment events for stage "2. design changes".

Our decentralised process model enactment approach has significant advantages over centralised approaches. Failure of any user's environment or a central server does not prevent other users from modelling and enacting their process models. As enactment events are recorded by environments, users rejoining the network can query these events, in the same manner they query for changes to process model views. In addition, large numbers of enactment events, which can occur if many users or many automated tasks are used, tend not to greatly reduce performance of our system as they are broadcast asynchronously and only to other relevant users.

One interesting consequence of our environment permitting asynchronous editing of process model views is that these can be inconsistent between different users' environments i.e. users may try to enact different versions of the same process model. This turns out to be generally unproblematic, as although environments may receive enactment events from engines running different versions of process models, these events may still be mapped onto the user's existing version stages, or shown to users if they cannot be mapped. We have found this approach allows users more freedom to evolve tailor the shared process models for their own purposes while still allowing them to monitor other users' work.

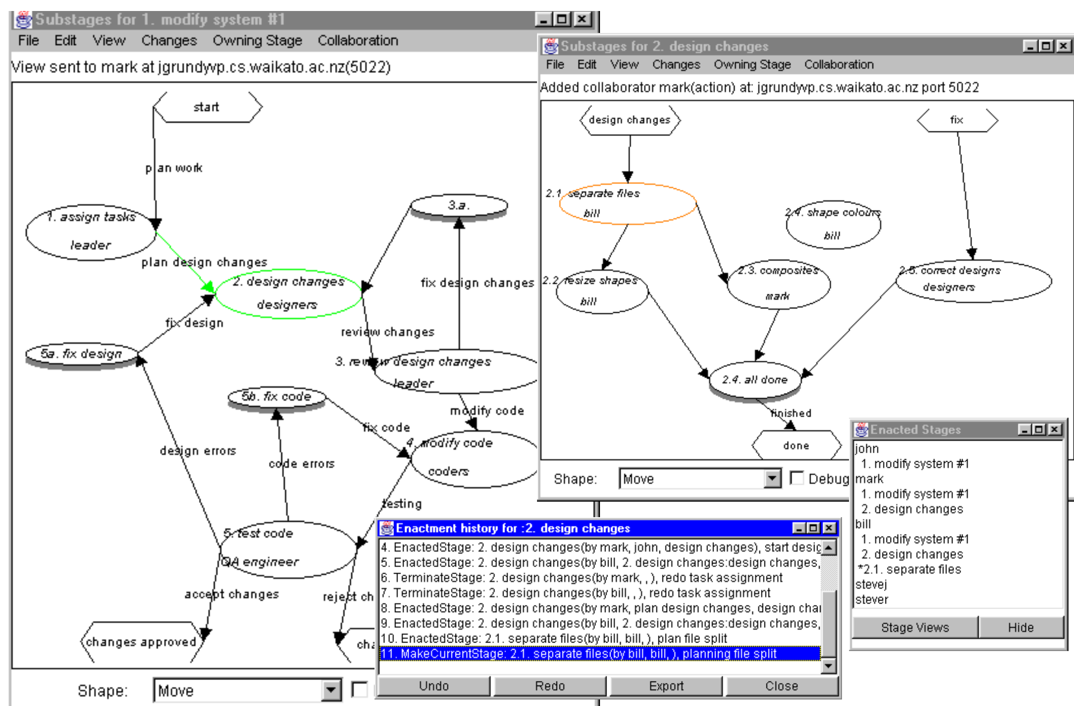


Figure 6. A simple example of an enacted process model.

6. Work Coordination Agents

While process modelling and enactment capabilities provide useful work coordination tools, they are by themselves of insufficient flexibility and power to assist users in managing complex cooperative software development work [8].

To address this, we have developed a novel visual language to support the specification of a range of software agents, including task automation, work history tracking, work coordination and tool integration, and to build extensions to the basic Serendipity-II process model behaviour. This event-based visual language comprises event "filters" (rectangular icons with a "?" prefix) and "actions" (oval, shaded icons).

Users connect event filters to process stages, roles, artefacts or tools, and these filters pass on any enactment, communication, artefact update or tool events matching specified criteria. Actions receive events, usually from filters, and perform specified processing in response. Figure 7 shows two simple examples of software agents specified by filters and actions. The model on the left detects when a stage is enacted or finished, and runs an action to download or upload artefacts relevant to the stage from a shared file server. The “request stage artefacts” and “put back stage artefacts” actions connect to the file server using sockets to transfer files (artefacts) associated in other views with the "2. design changes" stage. These actions are examples of packaged interfaces to third-party Information Systems (in this case a shared file server) being integrated with the process enactment engine. The model on the right defines an agent which detects changes made to a work artefact and stores these changes in an event history artefact (represented by the plain rectangular icon). Filter and action models can be packaged and parameterised by inputs and outputs, like subprocess models, and reused in different process model specifications.

One implication of our decentralised Serendipity-II architecture is that software agents can be run: locally for a user's environment only; in another user's environment; or by autonomous agents with their own "environment" (sender/receiver and uniquely identified object pool). Each approach is appropriate for different kinds of agent. For example, running agents locally is useful for basic task automation.

We have built agents to: add functionality to Serendipity-II, such as time delay enactment and automatic enactment; store selected changes to artefacts and process models; store selected stage enactment events; and interface to third-party systems. The latter include tools for software development (editors, compilers and CASE tools), office automation (word processors, spreadsheets, and databases) and communication (email, chat and audio).

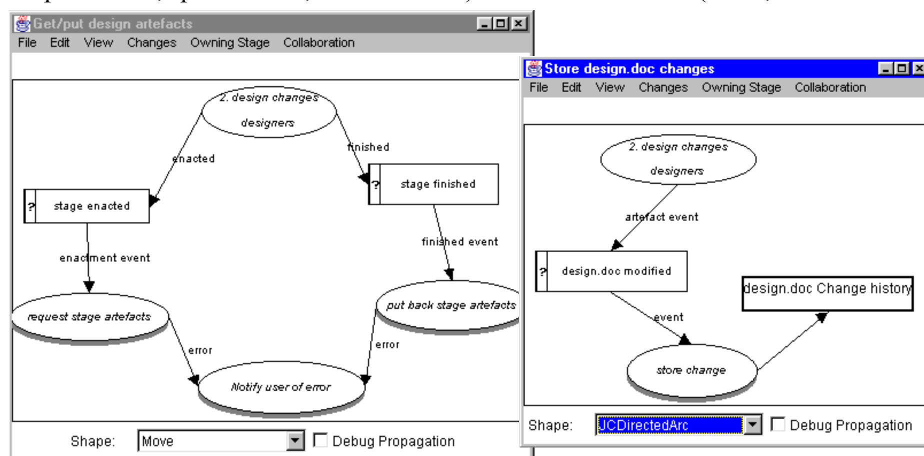


Figure 7. Two simple software agents specified in Serendipity-II.

Sometimes it is necessary for agents to process events generated in other users' environments. Such software agents must either request other environments send them such events (by asking other users to allow "event fowarding" agents to be added to their environments), or these software agents must be run entirely by the other users' environments. To achieve the latter, users specify the event processing they want with Serendipity-II filter and action views, then send these views to other users, who decide whether or not to allow the software agents specified in them to run in their environment. This allows users to ensure inappropriate agents which could adversely affect their work are not run within their environments. This was a major drawback with our orginal centralised Serendipity architecture: users lacked sufficient control over the agents others could specify which affected their work [8].

Autonomous agents are specified in the same way as other agents, and are sent to "robot" environments, which don't have a user interface but instead only run automatic processing. These robots run filter/action models sent to them, usually communicating with multiple users' environments to request appropriate events to process. These agents may continue to run when a user disconnects from the network. On reconnection, they can be queried for events that occurred during a user's absence or for results of processing. We have built several such autonomous agents, for example to provide a centralised work history tracking and querying facility, to embody constraints which affect multiple users' process models, and for additional group awareness capabilities. Our decentralised architecture can tolerate such agents failing and being restarted, whereas most process support systems using centralised approaches can not.

Figure 8 shows an example of a distributed software agent being specified by user "John". The left hand view is sent to user "Bill's" environment to forward changes of interest to John automatically. Bill must however first concur to have the components of this view added to his Serendipity-II repository. The right-hand view specifies that changes forwarded from Bill's environment are stored for later perusal by John (but these could, for example, be passed onto further filters and actions for automatic processing).

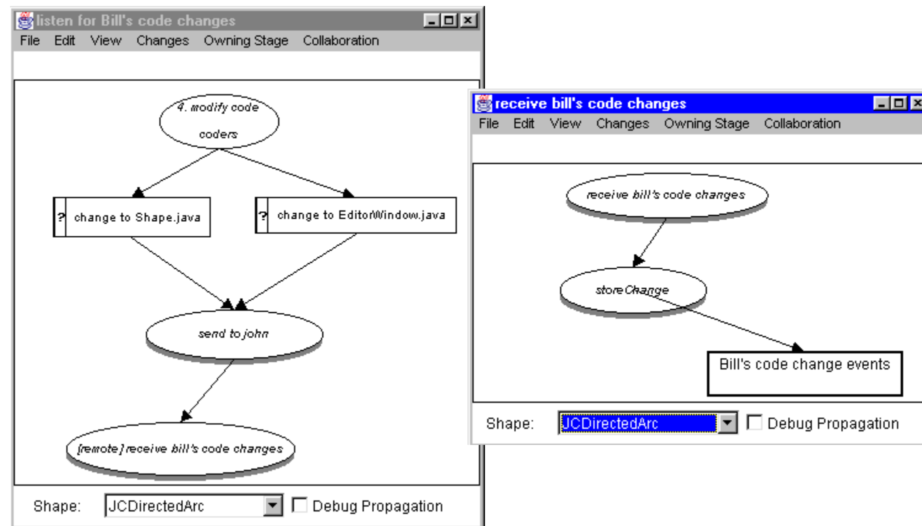


Figure 8. An example of work coordination using distributed filters and actions.

7. Discussion

Many WFMS based on centralised, client-server style architectures, such as Regatta [15], Action Workflow [13], SPADE [2], and ProcessWEAVER [5] suffer from performance problems when database access and message passing frequencies are high. This is particularly problematic where large numbers of developers need to coordinate work, and where many autonomous work coordination and task automation agents are employed. Such systems also suffer notorious robustness problems, with the central server being their weakest link [6]. In contrast, our decentralised architecture performs well even under heavy loading. This is due to the devolution of responsibility for inter-environment communication to each user's environment, with no central server bottlenecks. Users' environments communicate only with "interested" environments when collaborative process modelling. When enacting process models, initial enactment events are sent to others' environments, whose own enactment engines then action these events. Robustness is ensured as any user or agent can be removed from the network and the rest of the system still continues to function, with dynamic rerouting of messages (if necessary) possible. When a user's environment reestablishes its connections to the network, other users and agents are queried for changes to components and events that occurred when it was off-line. Our environments record all such events to ensure this is possible.

Various other decentralised approaches supporting workflow enactment have been developed, including those of METUFlow [6], Exotica/FMQM [1], and MOBILE [10]. Some of these, such as MOBILE and Exotica, use replicated servers, process migration and assignment of parts of workflows to specific servers. These require complex algorithms to assign workflow processing, and still retain servers, which may fail or become bottlenecks. METUFlow assigns workflow execution to CORBA objects, with computed guards controlling distributed execution, but requires a block-structured workflow language and a guard construction mechanism. Many of these approaches are not tolerant of periodic network disconnection for neither process modelling nor enactment. While much work has been done on distributed workflow enactment, most such systems do not support decentralised specifications of workflow models, but require client-server or single-user workflow definition [1, 13, 15], like most other collaborative editing tools. We have found our decentralised approach to both workflow modelling and enactment to be more flexible and robust. An additional advantage of Serendipity-II is its use of visual, event-based workflow modelling, enactment and software agent specification languages. Such languages tend to be more readily understandable by end-users than many textual, rule-based languages [5, 8, 15].

Various Computer-supported cooperative work (CSCW) systems have been developed to facilitate collaborative modelling and design. Examples include those with centralised architectures, such as GROVE [4] and ImagineDesk [3]. Those with decentralised architectures include GroupKit [14], Orbit [11] and Mjolner [12]. Generally, CSCW

tools with centralised architectures share the same performance and robustness limitations as centralised process support tools. Many decentralised CSCW systems, like GroupKit, also do not scale up for use on large, multiple view modelling domains, such as process modelling. This is often due to their adoption of universal synchronous or asynchronous editing modes for all users on different kinds of views. In Serendipity-II, we allow users to decide on the appropriate editing mode for each view, and which collaborators have access to the view and thus need be sent editing events. This minimises unnecessary editing event propagation.

Performance, robustness, security and distribution of Serendipity-II for process modelling and enactment has been very good in the software process modelling and office automation domains we have used it for. We have been deploying the environment on several small office automation process modelling applications, with some users on a local area network and others using mobile computers. We have also deployed it on a medium-size software process modelling and enactment application, with seven software developers and a variety of tools being used in conjunction with Serendipity-II. These include program editors and compilers, communicaion and file management services and JComposer being used as an OOA/D tool. Most developers are using a local area network, but two work from home using modem connections, and two are at another location 100km away. The internet provides a seamless, unifying communication mechanism between all environments enabling all developers to coordinate their work using Serendipity-II. Useability studies of Serendipity-II are underway and will help us further enhance the environment, and we plan to deploy it in other process modelling domains.

8. Summary

We have described the Serendipity-II process modelling and enactment environment and its decentralised architecture. This architecture uses multiple point-to-point communication across the internet between distributed users' and software agents, obviating the need for centralised servers. This results in more robust, efficient, secure and easily distributable process modelling and enactment and decentralisation of task automation and work coordination agents. Distributed process modelling allows users to join and leave a network of cooperating process modellers, with both asynchronous and synchronous process model view editing supported. Distributed process enactment is supported by broadcasting process enactment events between related users' environments, once again allowing users to join or leave the network of users and tolerating failure of any node in this network. Serendipity-II provides a visual process modelling language and a visual event-based software agent definition language. Software agents defined using event filters and actions can be run locally, or sent to be run in other users' environments, or by autonomous agent environments.

We are currently building further software agents to assist users in managing the complexities of distributed work. These include agents supporting document sharing, revision histories, interfaces to legacy development tools, and improved awareness of other users' work. Providing facilities allowing users to specify groups of other users to forward objects and events to, and to specify autonomous agent environment configurations, will allow end users of Serendipity-II to more easily configure their decentralised process support systems.

References

[1] Alonso, G., Mohan, C., Gunthor, R. amd Agrawal, D., ElAbbadi, A., and Kamath, M., "Exotica/FMQM: A Persistent Message-based Architecture for Distributed Workflow Management,"in *Proceedings of the IFIP WG8.1 Working Conference on Information Systems Development for Decentralized Organisations,* Norway, 1995.

[2] Bandinelli, S., DiNitto, E., and Fuggetta, A., "Supporting cooperation in the SPADE-1 environment," *IEEE Transactions on Software Engineering*, vol. 22, no. 12.

[3] DiNitto, E. and Fuggetta, A., "Integrating process technology and CSCW," in *Proceedings of IV European Workshop on Software Process Technology,* Lecture Notes in Computer Science, Springer-Verlag, Leiden, The Nederlands, April 1995.

[4] Ellis, C.A., Gibbs, S.J., and Rein, G.L., "Groupware: Some Issues and Experiences," *Communications of the ACM*, vol. 34, no. 1, 38-58, January 1991.

[5] Fernström, C., "ProcessWEAVER: Adding process support to UNIX," in *2nd International Conference on the Software Process: Continuous Software Process Improvement,* IEEE CS Press, Berlin, Germany, February 1993, pp. 12-26.

[6] Gokkoca, E., Altinel, M., Cingil, I., Tatbul, E.N., Koksal, P., and Dogac, A., "Design and implementation of a Disrtributed Workflow Enactment Service," in *Proceedings of 2nd IFCIS Conference on Cooperative Information Systems,* Charleston, SC, USA, June 1997.

[7] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., “Supporting flexible consistency management via discrete change description propagation,” *Software - Practice & Experience*, vol. 26, no. 9, 1053-1083, September 1996.

[8] Grundy, J.C. and Hosking, J.G., “Serendipity: integrated environment support for process modelling, enactment and work coordination,” *Automated Software Engineering*, vol. 5, no. 1, January 1998.

[9] Grundy, J.C., Hosking, J.G., and Mugridge, W.B., “Static and Dynamic Visualisation of Software Architectures for Component-based Systems,” in *Proceedings of SEKE'98*, KSI Press, San Francisco, June 18-20 1998.

[10] Heinl, P. and Schuster, H., “Towards a Highly Scaleable Architecture for Workflow Management Systems,” in *Proceedings of the 7th International Conference and Workshop on Database and Expert Systems,* Zurich, Switzerland, September 1996.

[11] Kaplan, S.M., Fitzpatrick, G., Mansfield, T., and Tolone, W.J., “Shooting into Orbit,” in *Proceedings of Oz-CSCW'96,* DSTC Technical Workshop Series, University of Queensland, Brisbane, Australia, August 1996, pp. 38-48.

[12] Magnusson, B., Asklund, U., and Minör, S., “Fine-grained Revision Control for Collaborative Software Development,” in *Proceedings of the 1993 ACM SIGSOFT Conference on Foundations of Software Engineering,* Los Angeles CA, December 1993, pp. 7-10.

[13] Medina-Mora, R., Winograd, T., Flores, R., and Flores, F., “The Action Workflow Approach to Workflow Management Technology,” in *Proceedings of CSCW'92,* ACM Press, 1992, pp. 281-288.

[14] Roseman, M. and Greenberg, S., “Building Real Time Groupware with GroupKit, A Groupware Toolkit,” *ACM Transactions on Computer-Human Interaction*, vol. 3, no. 1, 1-37, March 1996.

[15] Swenson, K.D., Maxwell, R.J., Matsumoto, T., Saghari, B., and Irwin, K., “A Business Process Environment Supporting Collaborative Planning,” *Journal of Collaborative Computing*, vol. 1, no. 1.