Preprint of paper published in International Journal of Software Engineering and Knowledge Engineering, © World Scientific 1999.

VISUAL SPECIFICATION AND MONITORING OF SOFTWARE AGENTS IN DECENTRALIZED PROCESS-CENTRED ENVIRONMENTS

JOHN GRUNDY

Department of Computer Science, University of Waikato, Private Bag 3105
Hamilton, New Zealand

Submitted 23rd January 1998
Frist Revision 29th May 1998
Accepted 3rd November 1998

Distributed, cooperating software agents are useful in many problem domains, such as task automation and work coordination in process-centered environments. We describe a visual language for specifying such software agents, which uses the composition of event-based software components. These specifications may contain interfaces to remotely executing agents, and agents may be run locally or on distributed machines using a decentralized software architecture. As facilities to configure and monitor the state and activities of such distributed, cooperating software agents is essential, we provide primarily visual capabilities to achieve this. Our static and dynamic software agent visualization techniques have been used on several projects where distributed information processing, system interfacing, work coordination and task automation are required. We illustrate our visualization techniques with examples from these domains.

Keywords: distributed software agents, visual languages, end-user computing, process-centered environments, work coordination, task automation

1. Introduction

Software agents are units of functionality that are able to carry out simple (or possibly complex) tasks, as specified by agent owners, in an autonomous fashion. Usually the specifier of an agent doesn't necessarily need to interact with the agent as it performs its tasks. Such agents are useful for WWW searching, electronic commerce, office automation and software engineering [1, 2].

We have utilized this concept of software agents to good effect in the process modeling and enactment (also called "workflow management") systems that we have been developing [3, 4]. Agents can be configured to carry out simple or complex automation of tasks, coordinate work by multiple people on projects, and interface workflow system components to third party software systems. These agents are specified by composing small units of functionality with a visual language. When executing, the state of these agents can be examined by users, and their configurations modified as required. In our previous work we have built mainly local machine agents, with some

centralized, shared agents [3, 5, 6]. These agents suffered from difficulties in specifying and controlling agents used by multiple people, difficulties in coordinating agent activities, and the inability to produce robust agents that don't require a centralized, single server architecture.

In this paper we describe a generalization of our work to facilitate the specification and deployment of distributed, cooperating software agents for a decentralized workflow management system. Our distributed software agents are specified using a simple visual language that composes two basic units of functionality. Event filters receive events from workflow components or other agents and pass these on to related agents if they conform to specified patterns. Actions receive events, typically from filters, and carry out some specified processing in response to the events received. Allowing such agent specifications to be distributed requires the use of intermediate agents to facilitate inter-agent communication and coordination. We describe visual language-based approaches we have developed to do this. Distributed agents need to be created, configured, inspected and monitored by users, or by other agents. We describe how this is achieved in our workflow system by extensions of the visual languages used to specify the agents. We then describe our experiences in deploying our distributed agents, compare and contrast our approach to related research, and describe improvements and generalizations of our approaches we plan to make in the future.

2. Problem Domain

Fig. 1 shows a screen dump from the Serendipity-II process modeling and enactment environment [4]. Serendipity provides visual languages for specifying process models and also utilizes these visual languages to provide information about enacted process models (stage enacted, who enacted etc). Fig. 1 shows part of a simple software process model (based on the ISPW6 Software Process Example [7]) modeled and enacted in Serendipity-II. The left-hand view shows a basic sub-process for modifying a software system, with ovals denoting *process stages*, lines denoting *enactment event* flows between stages, and hexagonal icons show *start/end states* of a sub-process model. In this sub-process, stages 1, 2 and 4 are enacted in order, indicating the project team plan, design and make code changes sequentially. The process stage "5. test changes" involves the project leader testing the code, and either indicating the design needs further changes, or accepting or rejecting the modifications. As stages are completed or suspended by users, enactment events flow along links between stages and in/out of sub-process models, driving process enactment. This is a generic, reusable software process model.

The top, right view shows a particular project sub-plan for the generic process stage "2. design changes". For this project, user John is the leader, and users Mark and Bill designers. Bill designs basic file changes first, then Mark and Bill concurrently make further design changes. This sub-process completes when both Bill and Mark have finished their changes.

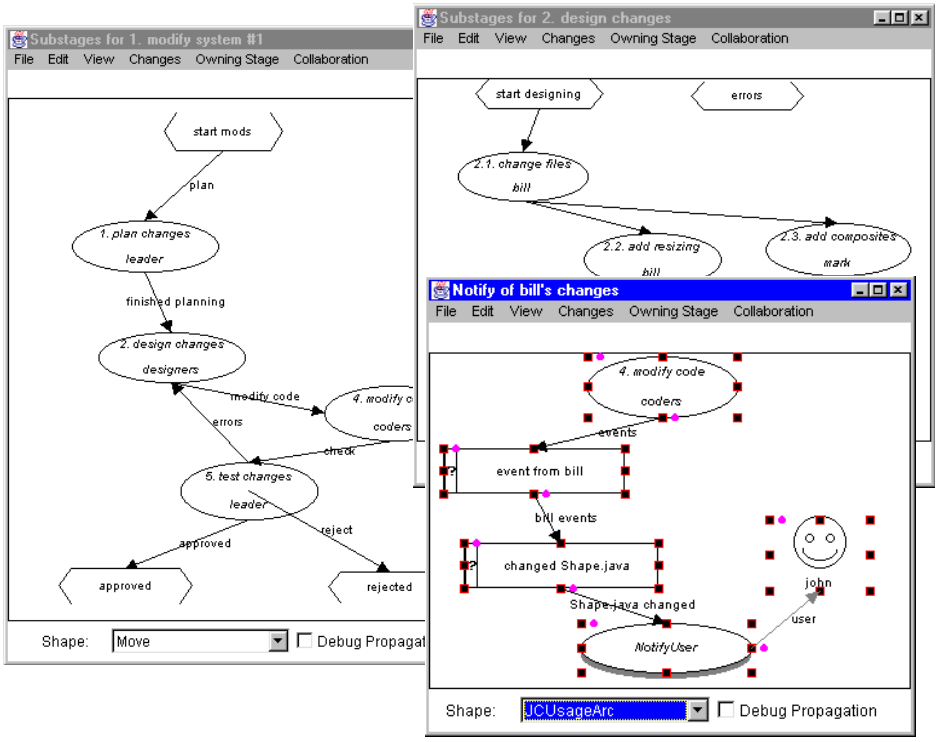


Fig. 1. The Serendipity-II process modeling and enactment environment.

In order to make more effective use of this process modeling and enactment facilities on projects, Serendipity-II needs to provide users with facilities to automate tedious and repetitive tasks [4]. For example, users often wish to be informed when another users accesses and/or modifies work artifacts they have an interest in, users often wish to record artifact modification and process enactment events so they can review work done on parts of a project, users need to have Serendipity-II interact with third party tools (e.g. detect events from them, invoke them, etc.), and need to collaborate not only on doing work but in using and evolving their shared process models. To facilitate these activities, we have added "software agents" to Serendipity-II, that are specified and deployed by end-users. These agents utilize a simple event filtering and actioning model, described by visual language extensions to the basic Serendipity-II process modeling and planning notation.

The bottom, right view in Fig. 1. shows a simple software agent that detects changes to file Design.doc by Bill and informs John of this. The rectangular icon is an *event filter* that receives enactment, tool or artifact change events from process stages or other filters. If the event matches a specified pattern, it is passed onto connected filters or actions. *Event actions* are denoted by shaded oval icons with one label. The "NotifyUser" action in Fig. 1. sends an email or chat message to the specified user, in this case John, denoted by the face icon. The arc connecting the action and role (face

icon) is a *usage connection*, in contrast to an event flow, indicating parameterization or a method calling relationship between the agent components. Event filters thus detect events of interest, while actions perform predefined or parameterized processing in response to received events. Process stages, in contrast, represent process model and enactment information.

While local (i.e. single-user) agents are useful, many agents must be distributed i.e. used by multiple users, and must be able to communicate and coordinate with other agents, both local and distributed. Our end-user agent specification and configuration facilities, and our software architecture for Serendipity-II, thus need to support agent distribution and interoperation. We do not use transparent locality of agents in Serendipity-II, as users need to ensure remote agents specified by others do not interfere with their process model and information artifacts. Thus users must always explicitly allow remote agents to have access to their events and/or process and local agent information, to ensure the security and integrity of their data.

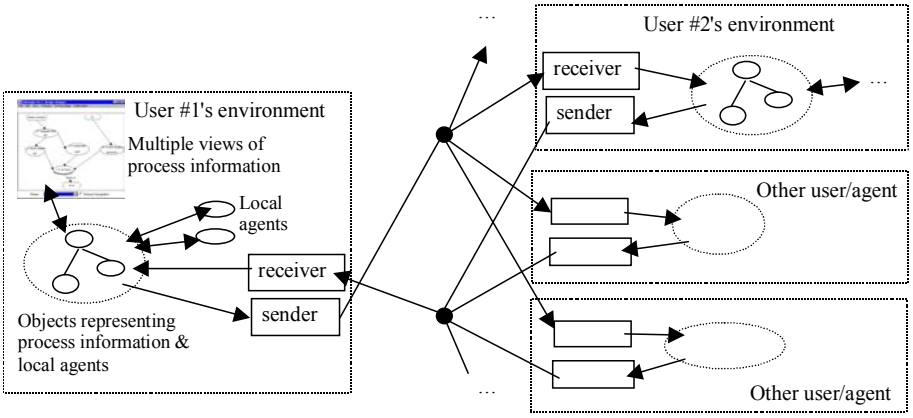


Fig. 2. The decentralized software architecture of Serendipity-II.

We have utilized a fully decentralized software architecture for Serendipity-II, in order to ensure its process modeling and enactment facilities are fast, robust and secure [4]. Fig. 2 illustrates this architecture. Each user has a collection of objects with identifiers unique to all Serendipity-II users, and these are used to model Serendipity-II repositories and multiple views. Objects can be replicated locally and remotely, and retain an indication of the object they were copied from. Each Serendipity-II environment provides object and event "sender" and "receiver" components. These allow the environment to broadcast objects and events to one or more other users' environments in a point-to-point fashion, and to receive objects and events from one or more other users' environments. We use a partially replicated architecture in which shared data is replicated on in users' environment, with events or replacement objects used to keep data consistent. This allows users to continue working even when others' environments

become inaccessible, and changes to object states which have occurred while a user is "off-line" to be later reconciled with other users' versions of process models, views and so on. Users may also have local objects which represent information "private" to that user. A user determines which other users can have copies of a subset of their objects, ensuring security of information access and usage.

Some environments in this architecture may, in fact, be autonomous software agent environments, whose operation is configured remotely by users via the sending of agent specifications to be run. Software agent specifications may also be run by a user's own Serendipity-II environment, as solely local agents or as agents that communicate with other, distributed agents.

3. Specifying Software Agents

We use a simple visual language to specify software agents. This language uses two main components: filters and actions. Filters receive events from other Serendipity-II components (other filters, actions, process stages, artifact interfaces or 3rd party software tools) and pass them on to connected components (usually other filters or actions). Filters may also provide a "querying" facility, where another filter or an action sends the filter a message and the filter replies to satisfy the query. Actions receive events, typically from filters although also from other kinds of Serendipity-II components, and carry out some specified processing in response to the event. Action processing may also be invoked in the same manner as querying filters, with the action returning a result in response to the query from a filter or other action.

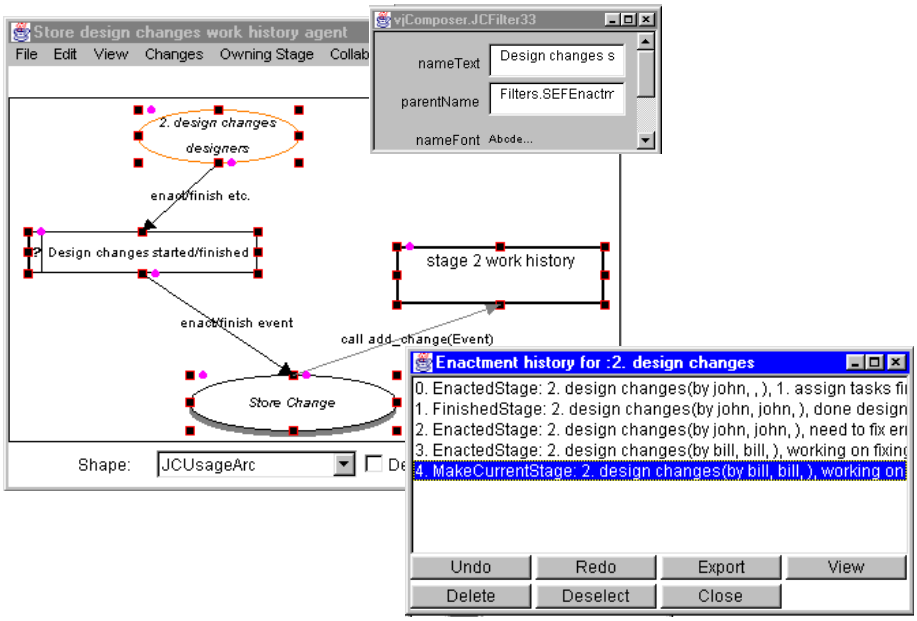


Fig. 3. Simple, local software agent specification in Serendipity-II.

Fig. 3 shows an example of a simple software agent specification in Serendipity-II. The process stage "2. design changes" produces "enactment events" when it is started, suspended, completed or the user indicates they are currently doing work on this part of a work process. This "Design changes started/finished" agent (which is reuses a predefined simple event filter) checks the event is a start or finish enactment event. If the filter passes this event on, the action "Store Change" records this enactment event in a Serendipity-II "version record" (event history) artifact, by sending the version record artifact component "stage 2 work history" an appropriate message. The reason a message send to the version record component is used, rather than propagating the event to it, is that version records don't respond to events - they provide methods to manage stored events. Thus filters and actions utilize a combination of event detection, response and propagation and message sending (i.e. method calling) as appropriate. The top dialogue shows the "property sheet" for the filter component, indicating the predefined filter it is reusing, while the bottom dialogue shows the work history version record's dialogue, displaying the events which have been recorded by this simple software agent.

Fig. 4 illustrates how the simple enactment event storage agent in Fig. 3 can be parameterized and reused. Two inputs (Stage and Storage) allow different process stages and event storage components to be utilized with this packaged software agent. Additionally, two outputs pass on the stored event (EventOut) or pass an "error" event if storage failed for some reason (ErrorOut). Other software agents can listen for such output events generated by this storage agent.

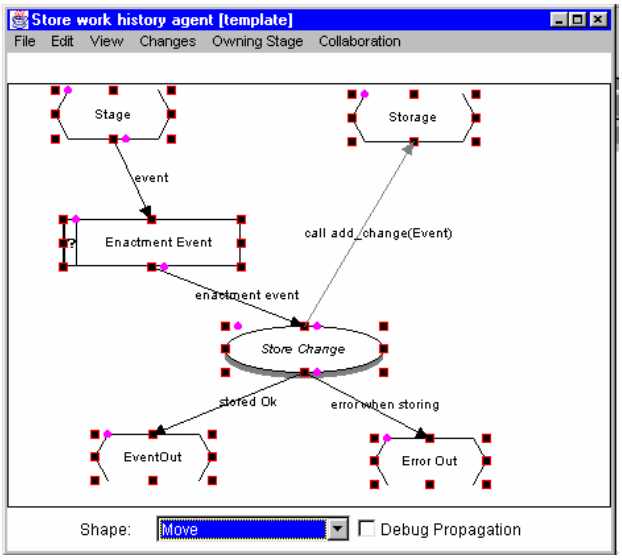


Fig. 4. Reusable, parameterized agent specification in Serendipity-II.

A wide variety of simple filters and actions are provided by Serendipity-II in a searchable component library. Users can build up their own software agents from these

simple filters and actions, and package these for reuse using appropriate parameterization, as illustrated in Fig. 4. New filters and actions can also be implemented in Java using an API, and added to the Serendipity-II library to dynamically extend the environment's capabilities.

4. Specifying Distributed Software Agents

In general, many software agents used with Serendipity-II need to be distributed i.e. be run by different users' environments, and some of these agents will need to communicate with other agents being run by different environments. Thus the specification of agents must include the ability to specify agents comprising of local and remote filter and actions. In Serendipity-II we have chosen to not make locality of agent components or agents themselves transparent to ensure users always know whether or not an agent is local and thus controlled by them, or if it is remote and controlled by someone else. This is to ensure that other users cannot specify agents that adversely effect a user's process models and local agent information, ensuring security and integrity of a user's environment. This was a major problem with its predecessor, Serendipity [3, 8], which used a centralized agent architecture and provided inadequate controls over other users deployment of agents.

We have extended the agent specification facilities of Serendipity-II to support the description of inter-agent communication, data sharing and coordination. Various issues arise when extending the specification of software agents in Serendipity-II to allow distribution of agents:

- Distributed agents or agent components need to be described i.e. which agents run as part of the local user's environment, which run on other machines, which machines do they run on, etc.
- Shared access to distributed data needs to be synchronized i.e. concurrent updates serialized or locking used on shared data components.
- Events need to be propagated to distributed filters and actions, messages sent to distributed filters and actions, and replies to messages received from distributed filters and actions. Some event and message propagation could be non-blocking, while others require the agent to which the request is sent to complete its response/processing before the sending agent continues execution.
- Distributed agents need some form of unique identification, so that other agents can identify and communicate with them.
- Distributed agents may become unavailable for periods of time, and agents should take this into account to ensure robust behavior when other agents fail

We utilize a variety of approaches to satisfy these requirements when providing distributed software agents in Serendipity-II:

- Local agents and remote agents can be specified in multiple views, with some views specifying local agents and other views remote agents
- Actions are provided that support remote event and message sending and receiving facilities
- Annotations can be used to indicate "remote" filters and actions by indicating the user or autonomous agent environment they are run by
- Actions are provided to ensure exclusive access to remote agents for periods of time, to serialize message and event propagation to shared, distributed agents, and to store events when remote agents are unavailable for retransmission at a later time

Fig. 5 shows an example of simple, distributed agent specification in Serendipity-II. In this example, a user "John" specifies two software agents using multiple views. One agent runs remotely as part of user "Bill"'s environment (the right hand specification), and detects events of interest to John. It then forwards these events to an agent in John's environment via a remote event sending action ("send to john's receiver"). The second agent runs locally on John's machine and receives changes from the remote agent, storing them for later perusal by John. If the event is a change to the artifact "Shape.java", then John is immediately informed by a broadcast message, by the action Notify by Message. The two attribute dialogues show attributes of the sender and receiver actions, publicizing the receiver component and telling the sender component which machine/receiver to send changes to.

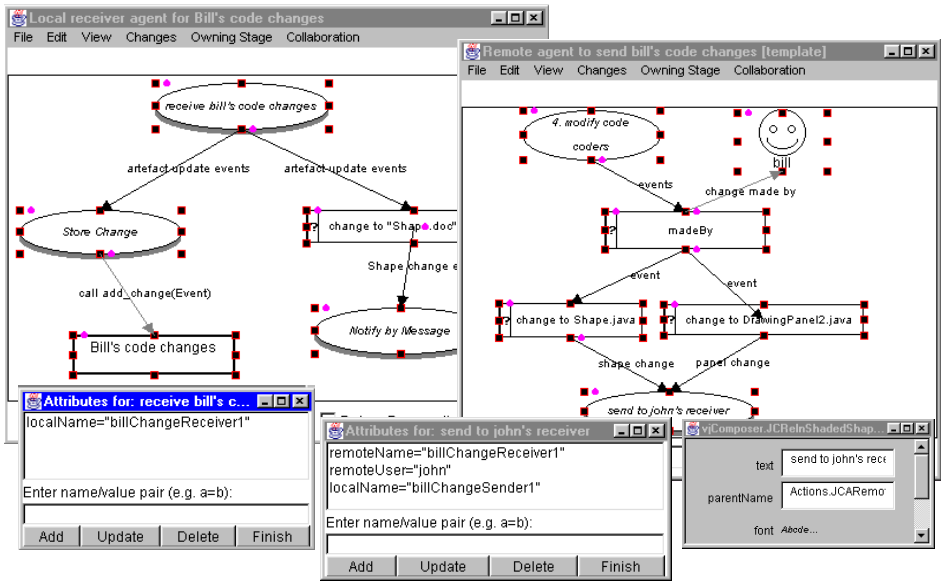


Fig. 5. Specifying distributed agents with multiple views and remote interfaces.

In this example, John defines the agent specification he requires, specifying both local (left hand view) and remote (right hand view) parts. He specifies properties of the remote agent interface action, including *remoteUser* and *remoteName*. When this remote agent interface action is run on Bill's machine, it forwards events it receives from the two filters to the specified host and local interface. In this example, these are John's environment and the local receiving interface action "billChangeReceiver1" - shown in the left hand view by action icon "receive bill's code changes". When this local event receiving action is sent events, it forwards these to the action and filter connected to it.

John cannot have the remote part of this agent automatically run by Bill's environment, but instead must ask Bill to run the event detection and forwarding agent in Bill's environment. This ensures users cannot run inappropriate agents affecting other users without their knowledge. The remote event sending action used in this example can be configured to store changes locally if the receiving agent is not running , i.e. if John is off-line temporarily. Such event sending agents can also be run by several other users, to allow John to be informed of events generated within several other environments. Events sent by remote sending agents are received by receiver environment receiver components, and are processed in serial by appropriate change receiver agents. These receiving agents can perform further processing of the received event e.g. order the stored events by sender, timestamp, etc. Serendipity-II provides a variety of other reusable filters and actions for distributed agent specification. These include priority queuing of messages and events sent to remote agents, locking of remote artifacts for read-only and/or exclusive access, time delayed agent execution, and so on.

We have chosen not to distinguish a local agent component from a remote interface in our notation, instead using the notion of reusable sender/receiver actions that handle remote data, message and event communication. This approach has proven to be a natural way of extending local agents to support distributed processing, and allows for a wide variety of actions supporting remote interfaces to be used.

Interfaces to remote agents can be encapsulated into other, reusable actions. For example, Fig. 6 shows the definition and use of an agent for communicating with a remote shared file repository. In this example, the agent definition is in the left-hand window, and specifies that when a stage enactment event is received by the agent, it will retrieve all files associated with the stage from the remote file server agent, locking these files for read-only by other users. The LockSharedFile and GetSharedFile actions provide a remote interface (in this example via a socket connection) with a file repository server. These could be replaced by actions which interface to a different shared file system/server, by actions which provide compatible interfaces for the GetSharedFiles action to use (i.e. another action that provides lockFile() and getFile() message interfaces).

This shared file check-out agent is reused in another agent specification in the right-hand view, to facilitate automatic check-out and check-in of design artifacts for a software process model. Note the use of an action to notify the environment user of

failure (e.g. remote file server agent down, error during data transfer, remote files locked by another user etc.).

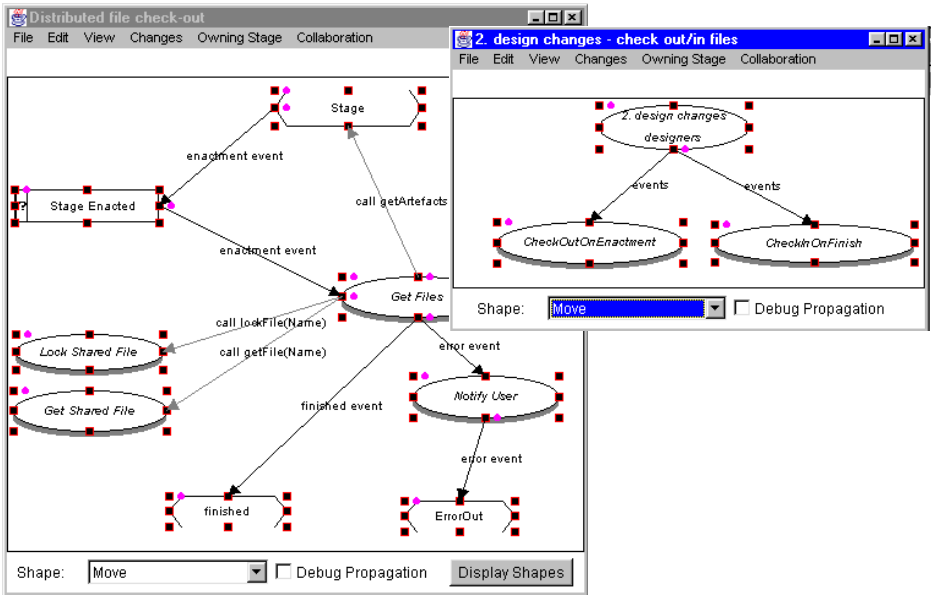


Fig. 6. Examples of remote system integration and task automation.

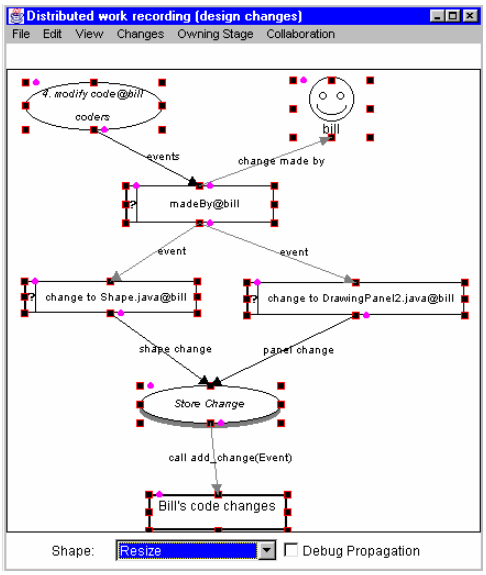


Fig. 7. Specifying distributed software agents with user discrimination annotations.

While the approach of using "remote" filters and actions in multiple view agent specifications works well, we have found that sometimes users wish to specify distributed agents with local and remote agent parts in the same view. Fig. 7 shows an example of this for the storage of enactment events generated by other users. In this example, user John has defined an agent that has components that run remotely. The names of these remote filters and actions are suffixed by a "user discrimination" annotation, which is used to determine which environment part of an agent should run on. Serendipity-II takes such a specification and generates local and remote agent specifications, and uses remote message and event propagation actions to facilitate their communication.

5. Using Distributed Software Agents

In order to use distributed software agents that have been specified as in the previous section, users must be provided with facilities to:

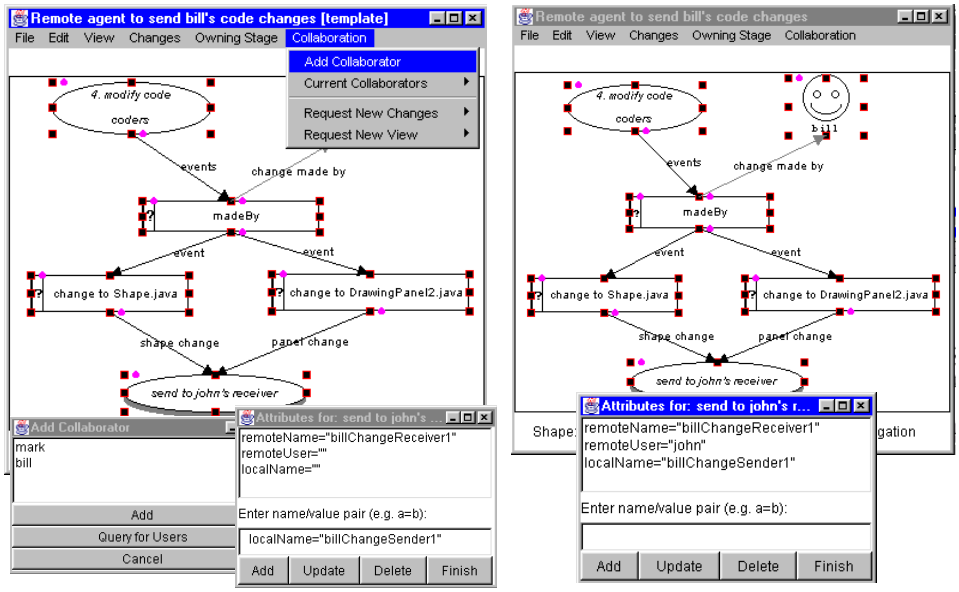
- create instances of agent specifications to be "run" in appropriate environments
- configure running agents e.g. connect to them other agent instances, disconnect them from other agent instances and set any parameter values they may have
- monitor and inspect running agent states and interconnections
- monitor running agent tasks and inter-agent coordination mechanisms

Local agent specifications are "run" as the user constructs the specification i.e. instances of appropriate filter and action components are created as the user builds the specification. When an agent receives events, it processes these according to its specification. Even partially constructed agents are executing, allowing users to incrementally build, test and refine their agent specifications. Agent specifications may be abstracted into templates for reuse. Such a template specification can not itself be run, but a copy of the template made, resulting in a running instance of the template agent. Serendipity-II provides facilities allowing users to modify agents copied from templates and to merge such changes back into the template. Similarly, if a template is modified, users can request such changes be applied to agents copied from the template definition.

Agents that are to be run remotely are defined as templates in a user's local environment, then sent to remote environment(s) for instantiation. Fig. 8 shows such an example of creating and configuring a distributed agent in Serendipity-II. In this example, John has created a template agent specification (the detect and send enactment event agent), and has set some parameters for this agent (these are attributes belonging to filters and actions that make up the agent specification which require values).

John has then asked Serendipity-II to send the configured agent template specification to user Bill's environment, by adding Bill as a collaborating user of this view. Bill has this agent specification imported into his environment as a template. This agent will not begin executing until Bill creates an instance of the template (i.e. authorizes the agent to run), which is illustrated in Bill's view in Fig. 8. If the agent is to

be run by an autonomous agent environment, the agent will be created and run if the user sending the template agent specification has authorization to have agents run by the receiving environment. When a user changes a remote agent's specification, changes to the template specification are sent to the remote user's environment for authorization before the remote agent is modified.



John's View Bill's View

Fig. 8. An example of creating and configuring distributed software agents.

Users may need to inspect and monitor running agent states, particularly when they are developing agents and wish to check agents are performing as expected. Users may also need to add new connections to running agents to add extra functionality to an environment. We allow users to inspect and reconfigure remote agent states by representing remote agents with environment discrimination annotations. Serendipity-II requests state information from these remote agents to present to users, and updates this information semi-synchronously as the remote agent state. Users can also specify new links from remote agents to local agents or to other remote agents, to reconfigure agent behavior or to monitor remote agent states (by displaying agent events/activity in version record dialogues, detecting "interesting" sequences of remote agent events and informing the user of these, etc.).

Fig. 9 shows an example of remote agent monitoring and reconfiguration. John is running an order entry/inventory management process model. An agent "Enact Reorder Products" detects when a product needs reordering, based on a lower threshold for the product in stock, and automatically enacts a reorder sub-process when this is required.

User Bill has defined an agent to monitor the performance and state of this remote agent of John's. User Bill has also opened the "property sheet" of this remote agent, as shown in Fig. 9. Serendipity-II requests the current state of the agent, and while this sheet is open asks the remote agent to send state change events so it can update the displayed state appropriately.

Bill has added two new links between John's remote agent and local software agents which are being used to monitor and display events entering the remote agent and "error" events generated by the remote agent. The addition of such links, and consequently remote change sender actions to forward events to Bill's environment, must be approved by the owner of the remote environment before the monitoring can begin. The "lb" annotation on the event flow into Store Events In action in this example denotes that events flowing INTO the remote agent are being forwarded, compared to events flowing out being sent to the filter.

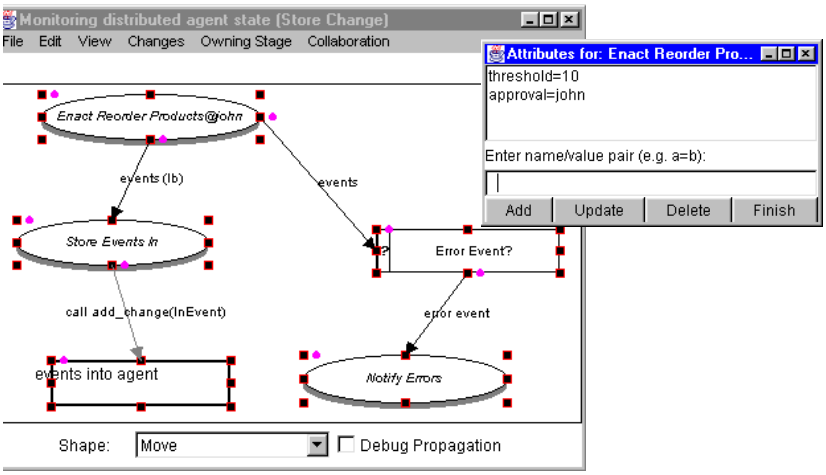


Fig. 9. Monitoring distributed software agent execution and state.

Distributed software agents in Serendipity-II often need to coordinate their work with other agents and users, need to keep track of tasks assigned to them and their progress on these tasks, and need to record information about the work they have done. All of these are achieved by defining shared Serendipity-II process models for agents, and allowing agents to distribute and enact these process models to plan, coordinate and record their work. Users may examine these enacted agent process models, if the agents make these accessible, to monitor the higher-level aspects of distributed agent behavior. Fig. 10 shows an example of distributed software agents for automating a simple inventory system management process model. The view on the left is the enacted process model used by these agents, which specifies the tasks the agents must perform, other agents and tasks they must coordinate with, and allows the agents to record the work they have done against these process steps. The views on the right are two

distributed agent templates, which run on autonomous agent environments. In this example, user John has requested the enacted workflow view and stage enactments, to examine what the agents have done to date. The enactment history dialogue shows the stage enactments and deenactments caused by the agents, recording their work to date.

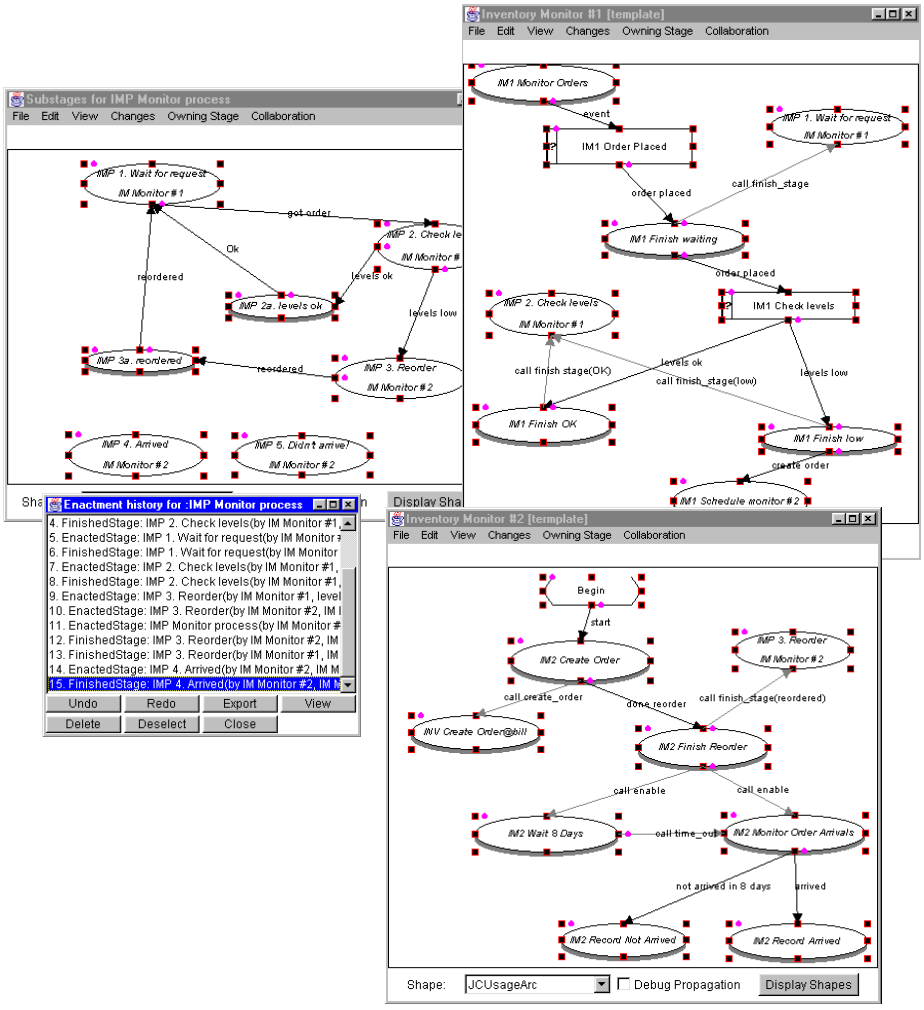


Fig. 10. Planning, coordinating, recording and monitoring agent work with process model views.

6. Implementation and Experience

Serendipity-II is implemented using JViews, a Java-based framework for building multiple-view, multiple-user distributed environments. A meta-CASE tool we have developed, JComposer, was used to design Serendipity-II visual languages and editors, and to generate the JViews classes to implement the Serendipity-II environment. JViews

utilizes Java Beans components, resulting in highly interoperable implementations, allowing environments like Serendipity-II to be integrated with a range of third party software systems. We utilized JViews abstractions to develop the decentralized architecture of Serendipity-II, and to develop a wide range of low-level filters and actions for distributed software agent specification. Performance of Serendipity-II with several users and a range of local and distributed software agents has proved more than adequate during its deployment to date. In part this is due to the multiple point-to-point agent communication mechanisms employed, and the use of autonomous agent environments to run various software agents.

The low-level filters and actions used to facilitate distributed software agent construction, such as remote change propagation and messaging, sequencing of event and message handling, locking of shared agents, and prioritization, were implemented in Java, using the JViews object serialization and communication framework abstractions. We built a range of reusable software agents on top of these using Serendipity-II itself, including agents for interfacing to other systems, remote event recording agents, and remote messaging and event propagation agents with various fault-tolerant behavior. We extended Serendipity-II itself so that filters and actions can be assigned to remote environments. Event and message links between these and other filters and actions running on local or other remote machines are translated by the environment into remote send/receive actions.

We have used Serendipity-II distributed software agents on a range of applications, including office automation and software development projects. To date we have found users of Serendipity-II generally find our visual agent specification and monitoring facilities easy to use for simple or moderately complex software agents. However, complex software agents require good command of these languages, and sometimes the underlying implementation of remote filters and actions, in order to achieve efficient and fully functional agents. We are currently generalizing the Serendipity-II remote filter and action components for use in JComposer itself. This will allow any JViews-based environment to define and utilize distributed software agents. Currently JComposer allows users to specify local agents using filters and actions similar to those of Serendipity-II, but does not include much support for distributed agent communication and coordination.

7. Discussion

A variety of approaches have been developed for visually specifying and monitoring distributed and parallel processing systems. Some examples include VISPER [9], HeNCE [10], ViTABaL [11], PEDS [12], Meander [13], Object Coordination Nets (OCNs) [14], and HyperReal [15]. Most of these systems are oriented towards general parallel and distributed systems programming, analysis and/or visualization. Examples of such systems include Meander, HyperReal, PEDS, and Hence. These languages thus provide a rich set of notational symbols and constructs for specifying parallel and distributed aspects of programs, inter-machine communication and inter-process

coordination mechanisms, locking and temporal sequencing. In our problem domain of providing tools for end-users to specify and monitor distributed software agents, many of these issues are handled transparently by our agent architecture. Some visual approaches for visual distributed systems specification, such as HyperReal, VISPER, ViTABaL and OCNs, could be utilized to specify distributed software agents in similar ways to Serendipity-II's visual agent programming language. These systems, however, also tend to incorporate a wide range of notational symbols and constructs to handle the complexities of general-purpose distributed systems specification. ViTABaL and OCNs match Serendipity-II's language characteristics the most closely, but deal with high-level distributed processing specification and interconnection, and rely on textual languages for more detailed process behavior specification. In contrast, Serendipity-II distributed agents can comprise visually specified local and distributed components, as appropriate, which utilize substantially similar notational conventions.

Various approaches have been developed for specifying autonomous software agents, including distributed agents. Examples include Zeus [16], AgentSheets [17], Agent Building Environment [18], JAFMAS [19], and Aglets [20]. Many of these approaches utilize purely textual languages or frameworks programmed in textual programming languages, and thus distributed agent development in such systems is time-consuming and mainly done by programmers. Some systems, such as AgentSheets, Zeus and Aglet Workbench's Tahiti, utilize visual languages to specify agent behavior and structure. Most such systems are rather limited in the range of agents that they can produce, with only agents with simplistic data, processing and interface characteristics being supported. Many agent-based systems provide mobile agents which can be highly distributed, but often do not allow developers to directly control agent inter-communication and coordination mechanisms. This results in inefficient or inappropriate agents, and sometimes means agent robustness is limited. While some systems, such as Zeus and Aglet Workbench's Fiji, provide a variety of agent monitoring and visualization capabilities, these are often limited to single agent states or histories, rather than visualizations of multiple agents states and histories of work and coordination.

Many workflow management systems and process-centered environments incorporate facilities for specifying aspects of software agents, many of which may function in a distributed fashion. Examples include SPADE [21], Oz [22], ProcessWEAVER [23], ADELE-TEMPO [24], Serendipity [3], and APEL [25]. Some systems, like ADELE and Oz, only provide textual, rule- or event-based languages for specifying software agent behavior. While these tend to be powerful and expressive, they give little high-level overview of agent interaction and structure, and are very difficult for end-users to understand and modify. Graphical approaches include petri net-based languages, like those of SPADE and ProcessWEAVER, which provide simple graphical overviews of structure but rely on (often complex) textual definition of transitions. More visual, event-based and dataflow-based approaches, like those of Serendipity and APEL, provide more accessible and expressive visual languages for end-users. However, most workflow and process-centered systems utilize centralized

databases and servers where agents are run, with very limited or no support for decentralized agents. Some workflow systems, such as METUFLow [26] and Exotica [27], provide decentralized architectures for workflow and agent execution. Unfortunately these systems do not use visual languages to support system definition and visualization, and are difficult to specify, configure and monitor with textual implementation languages.

The approach we have used in Serendipity-II to facilitate the specification and use of distributed software agents is to extend a visual, event-based language to incorporate a range of packaged components for use in distributed agents. Some limited extensions to the language have included annotations to indicate agent locality, but most components of the language are used for both local and distributed agent composition and visualization. This has great advantages in terms of keeping the visual notation simple and consistent, enables end users to understand and deploy distributed agents themselves, and many distributed, cooperating agents useful in our problem domain are easy and quick to build and deploy. The use of a highly decentralized architecture on which these visual agent specifications are run results in secure, robust agent implementations. We have successfully developed and deployed a range of agents using our techniques, and have had several diverse agents developed by non-programmer uses of Serendipity-II.

Due to our approach of packaging most facilities for building distributed agents into reusable filters and actions, users must determine and deploy appropriate agent communication and coordination mechanisms using appropriate filters and actions. In addition, while some distributed agent components we provide anticipate and handle distributed agent failure, many agents have such behavior built into them. This is not always straightforward in complex workflow systems, and relies on agent designers understanding and utilizing our architecture appropriately. Systems which handle such coordination, communication and failure detection transparently alleviate this sometimes difficult task, although are consequently less flexible. Designers of Serendipity-II agents can produce very inefficient agents if inappropriate aspects are distributed. For example, if an agent only acts upon a certain kind of event, having a remote stage or other agent send every event to the agent for filtering is very inefficient. Putting event filtering on the remote environment, and having it only forward appropriate events, is much more efficient but our system relies on agent designers to make such a decision. Analyzing and monitoring remote agent execution is still difficult in Serendipity-II, with improved facilities to visualize event and message propagation inside and between remote agents highly desirable.

We are currently enhancing the visualization capabilities of Serendipity-II to provide feedback to users on distributed agent behavior over time. To do this we are recording events sent to agents and propagated by agents, in addition to state information provided by agents. This information can be viewed in a variety of ways e.g. event traces, cause-effect graphing, animation of agent specifications to show execution etc. We are also developing a different kind of view for specifying agent locality and inter-agent

communication, using software architecture diagrams. Agents are assigned to environments with these diagrams, and communication mechanisms can be inferred from inter-environment links.

8. Summary

Distributed workflow management systems can make use of software agents to assist users to coordinate their tasks, automate repetitive tasks and integrate workflow systems with, possibly distributed, third party software tools. We have described a primarily visual language for specifying both local and distributed software agents for a workflow system which utilizes a decentralized architecture. Agent specifications incorporate event filters and actions which may facilitate communication and coordination between distributed agents. Agents may additionally have annotations indicating machine locality, with inter-machine communication mechanisms generated between local and remote agents. Running distributed agent states may be visualized, and their entry and exit events intercepted for monitoring purposes. Agents can themselves utilize enacted workflow models to coordinate and record their work. We have utilized our distributed agent specification and monitoring techniques in several small-to-medium scale application domains, and are continuing to enhance our languages and components for distributed agent construction and visualization. Our experiences with these applications has demonstrated that visual language approaches to specifying and monitoring distributed software agents are useful techniques in this problem domain.

Acknowledgements

The helpful comments of the anonymous reviewers on an earlier draft of this paper are gratefully acknowledged. Funding support was provided for part of this research from the New Zealand Public Good Science Fund.

References

1. H.S. Nwana, Software Agents: An Overview. *The Knowledge Engineering Review* **11** (1996) 205-244.
2. D.T. Ndumu and H.S. Nwana, Research and development challenges for agent-based systems. *IEE Proc. on Software Engineering* **14** (1997) 2-10.
3. J.C. Grundy and J.G. Hosking, Serendipity: integrated environment support for process modelling, enactment and work coordination. *Automated Software Engineering* **5** (1998) 27-60.
4. J.C. Grundy, J.G. Hosking, W.B. Mugridge and Apperley, M.D. A decentralized architecture for process modelling and enactment, Internet Computing 2 (1998), IEEE CS Press, 53-62.

5. J.C. Grundy, W.B. Mugridge and J.G. Hosking, "Utilising knowledge of past events in Process-Centred Software Engineering Environments", in *Proc. of 1997 Australian Software Engineering Conference,* Sydney, Australia, (Sept./Oct. 1997), IEEE CS Press, pp. 127-136.
6. J.C. Grundy, J.G. Hosking and W.B. Mugridge, "Low-level and high-level CSCW in the Serendipity process modelling environment", in *Proc. of 6th Australian Conf. on Computer-Human Interaction,* Hamilton, New Zealand (Nov. 1996), IEEE CS Press, pp. 69-77.
7. M.I. Kellner, P.H. Feiler, A. Finkelstein, T. Katayama, L.J. Osterweil, M.H. Penedo and H.D. Rombach, "Software Process Modeling Example Problem", in *Proc. of the 6th International Software Process Workshop,* Hokkaido, Japan (Oct. 1990), IEEE CS Press.
8. J.C. Grundy, J.G. Hosking and W.B. Mugridge, "Coordinating distributed software development projects with integrated process modelling and enactment environments", in *Proc. of 7th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises,*Stanford, CA (June 1998), IEEE CS Press, (in press).
9. N. Stankovic and K. Zhang, "Towards visual development of message-passing programs", in *Proc. of the 1997 IEEE Symp. on Visual Languages,* Capri, Italy (Sept. 1997), IEEE CS Press, pp. 144-151.
10. A.L. Beguelin, "HeNCE: Graphical Development Tools for Network-based Concurrent Computing", in *Proc. of the 1992 Scalable High Performance Computing Conf.,* Williamsburg, VA (April 1992), pp. 129-136.
11. J.C. Grundy and J.G. Hosking, "ViTABaL: A Visual Language Supporting Design By Tool Abstraction", in *Proc. of the 1995 IEEE Symp. on Visual Languages,* Darmsdart, Germany (Sept. 1995), IEEE CS Press, pp. 53-60.
12. D.Q. Zhang and K.A. Zhang, "Visual Programming Environment for Distributed Systems", in *Proc. of the 1995 IEEE Sym. on Visual Languages,* Darmsdadt, Germany (Sept. 1995), IEEE CS Press, pp. 310-317.
13. G. Wirtz, "A Visual Approach for Developing, Understanding and Analyzing Parallel Programs", in *Proc. of the 1993 IEEE Symp. on Visual Languages,* Bergen, Norway (Sept. 1993), IEEE CS Press, pp. 261-266.
14. G. Wirtz, J. Graf and H. Giese, "Ruling the behaviour of distributed software systems", in *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications,* Las Vagas, Nevada (July 1997).
15. F. DePaoli and F. Tisato, "Architectural Abstractions for Real-Time Software", In *Proc. of the 2nd Asia-Pacific Conf. on Software Engineering,* Brisbane, Australia (Dec. 1995), IEEE CS Press, pp. 199-208.
16. H.S. Nwana, D.T. Ndumu, and L.C. Lee, "ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems", in *Proc. of 3rd Practical Application of Intelligent Agents and Multi-Agents,* London, UK (April 1998).
17. A. Repenning, "Bending the Rules: Steps toward Semantically enriched Graphical Rewrite Rules", in *Proc. of IEEE Symp. on Visual Languages,* , Darmstadt, Germany (Sept. 1995), IEEE CS Press, pp. 226-233.

18. D. Gilbert, “Intelligent Agents: The Right Information at the Right Time”, May 1998, http://www.networking.ibm.com/iag/iaghome.html.
19. D. Chauhan, “A Java-based Agent Framework for MultiAgent Systems Development and Implementation”, May 1998, http://www.ececs.uc.edu/~abaker/JAFMAS/JAFMAS.html.
20. D.B. Lange and D.T. Chang, “IBM Aglets Workbench: Programming mobile agents in Java,” September 1996, http://www.trl.ibm.co.jp/aglets/whitepaper.htm.
21. S. Bandinelli, E. DiNitto and A. Fuggetta, Supporting cooperation in the SPADE-1 environment. *IEEE Trans. on Software Engineering* **22** (1996), 841-865.
22. I.Z. Ben-Shaul, G.T. Heineman, S.S. Popovich, P.D. Skopp, A.Z. Tong and G. Valetto, "Integrating Groupware and Process Technologies in the Oz Environment", in *9th Int. Software Process Workshop,* Airlie, VA (Oct. 1994), IEEE CS Press, pp. 114-116.
23. C. Fernström, "ProcessWEAVER: Adding process support to UNIX", in *2nd Int. Conf. on the Software Process,* Berlin, Germany (Feb 1993), IEEE CS Press, pp. 12-26.
24. N. Belkhatir, J. Estublier and W.L. Melo, The Adele/Tempo Experience*, Software Process Modelling & Technology,* eds. A. Finkelstein, J. Kramer, J. and B. Nusibeh (Research Studies Press, 1994) 187-222.
25. S. Dami, J. Estublier and M. Amiour, APEL: A Graphical Yet Executable Formalism for Process Modelling, *Automated Software Engineering* **5** (1998), 61-96.
26. E. Gokkoca, M. Altinel, I. Cingil, E.N. Tatbul, P. Koksal and A. Dogac, "Design and implementation of a Disrtributed Workflow Enactment Service", in *Proc. of 2nd IFCIS Conf. on Cooperative Information Systems,* Charleston, SC (June 1997), IEEE CS Press.
27. G. Alonso, C. Mohan, R. Gunthor, D. Agrawal, A. ElAbbadi and M. Kamath, "Exotica/FMQM: A Persistent Message-based Architecture for Distributed Workflow Management", in *Proc. of the IFIP WG8.1 Working Conference on Information Systems Development for Decentralized Organisations,* Trondheim, Norway (Aug. 1995), Chapman & Hall.