In Knowledge-Based Systems, 1 (4), Sept. 1988.

1988b

Co-operation and communication within an active IPSE

Thomas Rodden, Peter Sawyer and Ian Sommerville

Most currently available Integrated Project Support Environments achieve integration of project components through the use of a cohesive set of tools layered on top of a database management system. In addition to the integration of tools and data, future IPSEs will be required to support integration of the various activities within a project. The ISM project is developing an architecture for a prototype IPSE which supports activity integration. The IPSE is implemented as a federation of intelligent, co-operating agents which communicate with each other and with IPSE users by message passing. This paper is particularly concerned with the mechanisms employed to permit inter-agent and agent-user communications, both to initiate actions within the IPSE and to enable the direct manipulation of the IPSE object store.

Keywords: knowledge base, user interaction, software tools, programming environment, IPSE

In common with any engineering task, effective software engineering requires the use of appropriate tools. In recognition of this fact, much effort has been invested in producing software tools such as compilers, symbolic debuggers, program analysers, etc. The concept of programming environments arose from the idea of collecting such tools together into kits where users could apply them at appropriate stages during the course of software development.

Unix[1] is perhaps the best known example of such a programming environment. The Unix system provides a variety of different tools, but more importantly provides tool interconnection mechanisms (i.e. character files, I/O redirection, pipes, shell programming) which allows tools to be used in concert. Outputs from one tool can serve as inputs to others, thus powerful tools and tool sequences can be built by putting relatively simple tools together.

In programming environments like Unix the software developer must take full responsibility for the application of the correct tools to the appropriate components of the design exercise. The environment itself encapsulates little information regarding interdependence of tools and components beyond that which can be expressed in Makefiles and shell scripts. Furthermore, the basic Unix toolset is principally intended for programming support. It does not provide a great deal of support for other software process activities.

A project support environment differs from a programming environment by providing support across a broad spectrum of project activities, from initial specification through to product maintenance. Only if that support is provided in such a way that the environment views the various tools, not in isolation, but as a set of interdependent activities forming part of a project *process*, it can be said to be an *Integrated* Project Support Environment (IPSE).

ASPECT[2] and ECLIPSE[3] are two examples of recent IPSE designs which address the problem of supporting the whole software development process with an integrated set of software tools. Both environments are typical of current IPSE designs: a tool set is layered around a project database, access to which is governed by an object management system. The object management system provides mechanisms for maintaining consistency among project components to a degree which was not possible in older, file-store based environments. A user interface based on bit-mapped workstations and direct manipulation is provided on both of these systems.

The current generation of IPSE systems, typified by ECLIPSE, may be termed 'passive' IPSEs, which exhibit two levels of integration:

1 Data integration via a database management system.
2 User interface integration via a consistent metaphor and standards.

They do not support activity integration. The activities involved in the software process are initiated entirely by user actions. In an active IPSE, as well as data and interface integration, activity integration is also supported. Information regarding interdependencies of tools and components may be used to initiate actions automatically when some conditions are met. These conditions may not necessarily be as a (direct) result of external stimuli.

Because of the relatively unstructured nature of the software process and the non-deterministic patterns of

Department of Computing, University of Lancaster, Bailrigg, Lancaster LA1 4YR, UK

0950–7051/88/040240–09 $03.00 © 1988 Butterworth & Co (Publishers) Ltd

activity activation, we have used techniques which have previously been applied in artificial intelligence (AI) and knowledge-based systems to the development of an active IPSE.

The IPSE contains a knowledge base, part of which includes a project model. This model encapsulates the knowledge (a project's, aims, products, resources, timescales, etc.) which was identified[4] as being necessary for tackling the problem of complexity in large software engineering projects. This information is embodied within the IPSE as objects. By manipulating these knowledge sources the IPSE is able to reason about an evolving project, integrate the various transformations which need to be applied to it's components, and thus automate many software process tasks. The ability to reason about a project and generate transformations automatically embodies the IPSE with the attribute of being *active*.

ISM: AN ACTIVE IPSE ARCHITECTURE

The work described here has been carried out in the context of a collaborative project called ISM[5]. The ISM project is a research programme started in October 1986 which is investigating the structure and applications of knowledge-based IPSE. Its main objective is to identify a suitable architecture and develop a prototype for such a system.

The project has concentrated on identifying a set of generic facilities integral to the design of an automated IPSE. In addition, it has focused on project management activities, including planning, configuration management and activity co-ordination. Applications of these facilities are being built to further explore specific areas in order to realize a prototype ISM, and an initial system was due for completion in August 1988. This paper describes the overall architecture which is being developed for ISM, and the facilities provided to support activity co-ordination, co-operative working and user interaction.

In the course of the prototype active IPSE design, techniques were utilized drawn from areas of AI where the technology is both appropriate and sufficiently mature. In most current environments, tools are layered around the IPSE kernel, where users interact with them. By contrast, our model of an active IPSE has neither tools nor a kernel in the accepted sense, but consists of a federation of co-operating agents (see Figure 1) which embody sufficient contextual knowledge to be invoked on an opportunistic basis.

An agent encapsulates both local data and the operations which may be performed by the agent. The external interface to an agent is formed by the set of its available operations. Operations on data held by an agent is normally performed by sending a message to the agent requesting somr action. Controlled access of data allows sharing of information required by a diversity of agents. There is a finite set of messages which may be accepted by any agent where each valid message is processed by the agent's corresponding operation. Agents may be automated or human.

The integral part played by human agents does not compromise the environment's claim to being active. If all IPSE agents, both software and human, are properly integrated, then they will all appear to the IPSE to have a consistent external interface. This enables the automatic scheduling of all agents to perform specific,

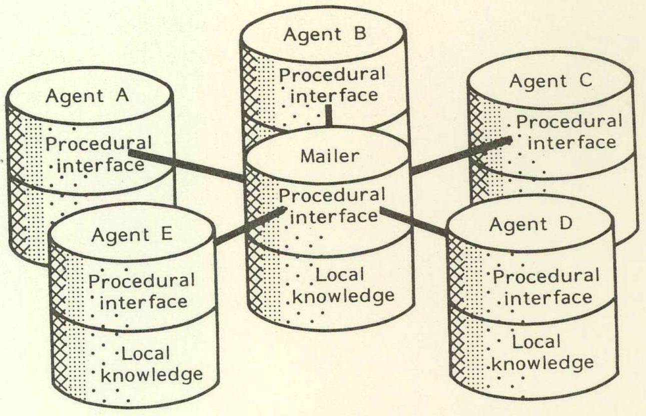


Figure 1. Logical view of active agent topology

project-related tasks, as black-box processes.

A powerful attribute of the IPSE, which is enabled by its integrated architecture, is the ability to delegate. Unlike most current environments, whose tools operate largely in isolation, many complex tasks can be performed automatically by agents co-operating. Thus, a complex operation may be disassembled into subtasks, each of which is delegated to the agent with the appropriate ability to perform the subtask, e.g. the design of an important program module may be running late. A consequence of this could be that the slippage will have to be absorbed by temporarily reallocating other members of the design team to work on the module. Finding an acceptable solution to this problem which minimizes the resulting knock-on effects will involve the co-operation of the project management agent, a planning agent, a scheduling agent, and one or more human agents.

The above example illustrates the requirement that agents must be asynchronous processes capable of running concurrently. The project management agent would not be able to cause the rescheduling of the project until the planning agent had established a preferred reallocation of resources (the execution time of a rescheduling process may be in the order of days if human agents are involved), yet it may be required to respond to other events during the intervening period.

In principle, any agent may call upon any other agent to perform a task by sending it a message. This implies that agents must embody information about the services offered by potential collaborators so that messages are not sent to agents which do not have appropriate behaviours with which to react. The approach which has been taken is to devolve this knowledge to an intelligent **mailer agent**.

An agent requiring a service from the IPSE sends a message via the mailer agent, which routes the message to the appropriate agent to respond to the request. In addition, the mailer agent allows users (human agents) to initiate actions within the IPSE in the same manner as any other agent, while also allowing other agents to request services of a user as they would of any agent. The mailer agent provides the user with a range of facilities to construct a personalized interface to the system, which buffers him from the abundance of messages common to actor-based systems[6,7]. Thus, as well as co-ordinating communications between active agents, the mailer agent can act as an electronic mail system.

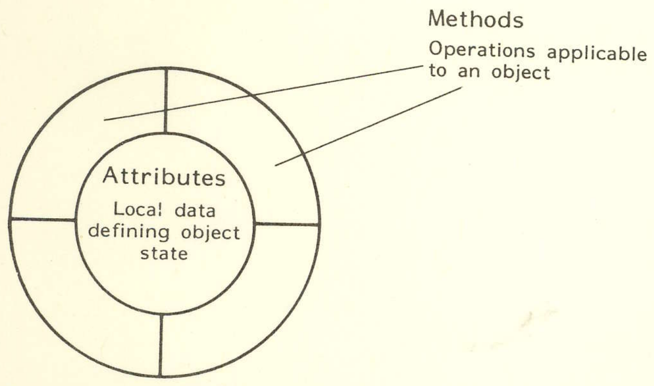

Methods
Operations applicable to an object
Attributes
Local data defining object state

Figure 2. Conceptual view of an object

In addition to being able to initiate actions within the IPSE by sending messages to agents via the mailer, a specialized user interface based on direct manipulation[8] permits users to access the contents of the knowledge base. Consistency of data within the knowledge base is enforced by the **object browser agent.** The object browser co-operates with the mailer to translate user requests into the message format appropriate to the agents responsible for the individual items of data being accessed. The principle of protecting data from unauthorized access is therefore enforced consistently for all agents, human and otherwise.

KNOWLEDGE REPRESENTATION

Agents are responsible for performing transformations of entities within the IPSE knowledge-base. The contents of the knowledge base are distributed among the set of agents. Knowledge is held within this partitioned knowledge base as **objects.** Object oriented programming techniques[9,10] were selected as an appropriate knowledge representation mechanism using the same rationale employed in the selection of agents to embody the active elements of the IPSE.

Object oriented programming

Object oriented programming enforces the principles of information hiding, encapsulation of data and processes, and provides a conceptually elegant means of packaging information. Using an object orientation methodology programs are structured as a collection of independant objects communicating via message passing. Objects encapsulate both **attributes,** data private to the object and *methods*, the operations which may be applied to the data.

Conceptually, objects are of the form shown in Figure 2. An object is a named **instance** of a **class** where a class can be thought of as a template definition of an object type. There may be multiple instances of any class, each of which represents a different project artifact of the same type, e.g. there may be an object class *project__meeting__minutes*, an instance of which is created after every project meeting.

A taxonomy of related object classes may be represented by an **inheritance tree.** An object class will inherit all attributes and methods defined for all its super classes. Thus, as a naive example, a program which models project resources may make use of the class *project__member*, for which the following attributes and methods are defined:

Attributes: Name; Grade; Location;
Methods: Assign.

A subclass, *programmer*, would inherit the attributes and methods defined for *project__member*, but may embody the additional definitions:

Attributes: Programming__skills; Current__assignment; Mail__address;
Methods: Send__on__training__course.

An instance of a class will have a state, defined by the values of its attributes. Consider the following instance of the class *programmer*:

Attributes:
Name: I.M.A. Hacker
Grade: Coder first class
Location: Room 21, MegaCorp House
Programming__skills: C, Ada, Unix, VMS
Current__assignment: EFA flight simulator
Mail__address: hacker@uk.co.megacorp.newsun

Within the IPSE knowledge base, attributes may have class default values, are typed, and may have constraints associated with them. Attribute constraints may serve not only to restrict possible attribute values as an extension to the typing scheme, but may also embody relationships between attributes and be capable of generating messages to other objects. Constraints applied to object attributes may be used as a powerful mechanism for inferring new attribute values from accumulated data, and for propagating knowledge across the system. In this respect, the object attribute constraint mechanism is similar to the idea of procedural attachment to frame slots[11].

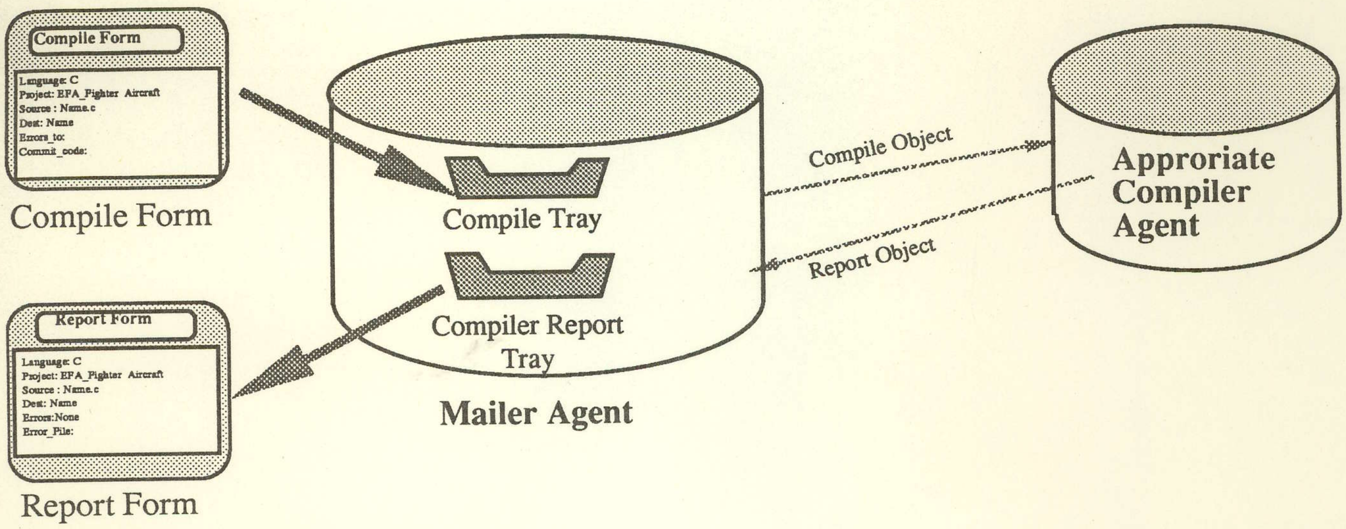
INTER-AGENT COMMUNICATION

User interaction style

A major area of research in recent years has been the design of user interfaces to computer systems. Traditionally, the most common style of interaction with a complex computer system has been via a command line interpreter such as that provided by the Unix c-shell. Command interpreters provide great power and flexibility and continue to enjoy popularity, particularly among those who have invested much time in gaining experience with their use. However, studies suggest that the complex command syntax endemic to this style of interaction can lead to considerable disorientation and confusion on the part of infrequent or inexpert users[12].

Modern computer workstations equipped with high resolution bit-mapped screens and pointing devices have now enabled alternative interaction styles to challenge the long standing supremacy of command line interpreters.

Interaction with ECLIPSE is performed via a user interface which uses a *control panel metaphor*[13]. The control panel metaphor is an example of the direct manipulation style of interaction in which data items are displayed continuously for the period of interest. Instead of having to know a complex command syntax, users perform physical actions like pressing a button, or selecting a menu item with a pointing device. When some operation has been carried out on an item, the effect is immediately visible to the user. The design of direct manipulation user interfaces are motivated by the



Compile Form

Language: C
Project: EFA_Fighter Aircraft
Source : Name.c
Dest: Name
Errors_to:
Commit_code:

Report Form

Language: C
Project: EFA_Fighter Aircraft
Source : Name.c
Dest: Name
Errors:None
Error_File:

Figure 3. Interaction via the mailer agent

goal of requiring users to carry less contextual knowledge around with them, freeing them to concentrate on the task-domain semantics.

Users interact with the active IPSE using a direct manipulation style of user interface based on form-filling. The following sections describe how this principle is used to initiate actions within the IPSE, and to manipulate objects within the knowledge base.

Mailer agent

The mailer agent controls all agent interaction and agent invocation within the IPSE framework. The sharing of knowledge is fundamental to any system which promotes co-operative working[14]. In addition, the co-ordination of interacting agents is a major consideration within a co-operative system such as the Active IPSE. A major role of the mailer agent is to aid in the realization of this co-operation.

The mailer agent's primary concern is how agents perceive their interaction with an active IPSE. Our implementation of an active IPSE is as a federation of co-operating intelligent agents with the ability to share a common information space, while utilizing message passing to collaborate on the disassembly and solution of various tasks. This architecture is similar to how users of electronic message systems perceive the mail community to which they belong[15].

By adopting this model agents may have a consistent interaction interface where exactly the same methods are used to send mail to system users, to file mail received and to initiate system actions. An agent initiates any action within the IPSE by passing an object to the appropriate agent, the conceptual view utilized by the mailer is the filling in and 'sending' of forms to other agent. Thus, for an agent to initiate the compilation of a program he fills in a *Compile* form (object). This form is then submitted to the mailer agent which *mails* the completed *Compile* form to the appropriate compilation agent (see Figure 3). By extending the message passing paradigm used within the IPSE to encompass the agent interface, the technologies and techniques applied within the message handling systems community become directly applicable to the realization of a cohesive interaction metaphor for the active IPSE.

However, while considerable research has been directed toward the development of efficient and reliable techniques for the transfer of messages, until recently comparatively little effort has been expended on interaction with message handling systems[7,16]. Message handling systems currently tackle many of the problems generated by communication within collaborative environments by providing an efficient and reliable communication medium. However, due to the historical emphasis on the transfer of messages, and the subsequent lack of development of user interface techniques, the presentation of information to the user tends to be rudimentary and completely unstructured, generating an effect which has been termed *Information Overload*[15].

When information overload occurs the flow of information is so rapid it makes it difficult for the user to discover information relevant to him. To be effective, systems must give message recipients the ability to discriminate between those messages they wish to read and those of little relevance to them[15]. Malone[17] conducted several studies of how various kinds of information are shared in organizations. The most interesting of the approaches he describes are cognitive filtering, where the decisions are based on the topic of the message, and social filtering, where decisions are based on who supplied the information. Additional studies[18] have shown that the majority of messages within electronic message systems are organizational, and a significant amount of these are routine in nature. We believe this will also be the case within an active IPSE.

Mailtray structuring

The structuring technique used is based upon the notion of *Mailtrays*, which forms flows into and out of as necessary[19]. Each mailtray has a title, a number of attributes, and an action list which describes how forms should be processed. Additionally, each mailtray has an associated guard list controlling the nature of the forms held in the mailtray (see Figure 4).

Mailtrays are active elements which accept or reject forms depending on their guard list. A tray's guard list is composed by its agent and describes the criteria for adding forms to the tray. The agent is free to create an interface reflecting its particular classification of forms, which may be fine tuned as required. Addition-

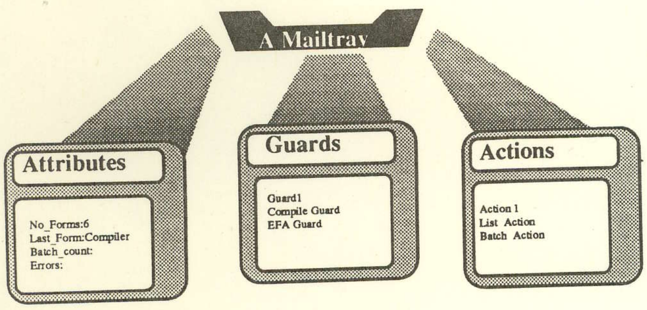


Figure 4. Mailtray components

ally, since mailtrays are dynamic in nature, an agent can amend this interface as requirements change.

Consider a member of a project team (a human agent) developing a component which interacts with various components developed by other team members. He may wish to arrange the various forms used so that all forms concerning program errors are grouped. He may also wish to collate all communication with any of the compiler agents used.

To do so he would define a number of trays in order that relevant forms are structured so that forms requiring immediate attention are dealt with directly, and routine forms are processed in as automated a manner as possible. Trays are defined by their creating agent using guards, which define what is placed in a tray, and an associated action list, which describes what should be done with the forms placed in a particular tray.

The decision whether or not a form is placed in a mailtray is controlled by the tray's **Guardlist**. The guardlist is a list of predicates which can be applied to the attributes of a form. If all the tests on a form's attributes succeed, the form passes that guard and is added to a tray's contents. Any number of guards may be associated with a mailtray, and a form is accepted if any of these guards is true.

The guard list for the tray defined to collate all forms regarding debugging might simply be:

Guard *all* **class** *debug*

This would allow all forms belonging to the class *debug* to enter the tray. However, if the user wished to collect compilation forms from a particular project, say spreadsheet, then he could alter the attribute list to allow only forms regarding spreadsheet to be added to the tray by replacing the guard above by one of the form:

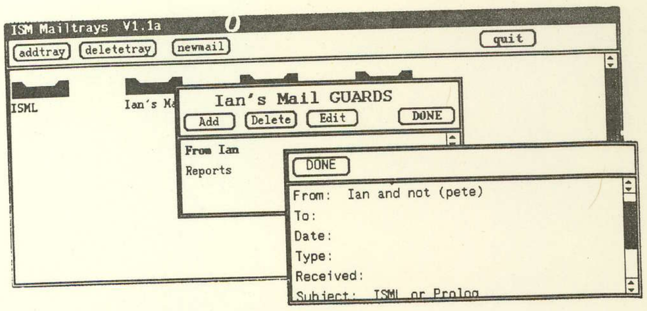
```
Guard all class debug
                [
                project = 'Spreadsheet'
                ];
```

For a form to successfully negotiate this guard it must be of class *debug* and have an attribute called project with a value 'Spreadsheet'.

When a form is placed in a mailtray the **Action List** for that tray is interpreted. Action lists consist of a list of commands to be executed. Each command is of the form:

Action Head → [Action Body];

The action head consists of the action name followed by the class of forms to which the action applies. If the form is of the class appearing in the action head then the succeeding action body is executed. Each action body is written in a notation consisting of *if then* and



From Ian
Reports

From: Ian and not (pete)
To:
Date:
Type:
Received:
Subject: ISML or Prolog

Figure 5. Mailtray interface

assignment statements in conjunction with form handling primitives (mail, save, forward, etc.), e.g. a user may wish to process forms of class *code test* only when he has ten forms of this type. His action list for the appropriate tray would contain the action *batch*:

```
batch : code__test → [
   no__forms = no__forms + 1;
   Save test__forms
   if no forms = 10
      then
          [
             Flag;
             no__forms = 0;
          ]
   ];
```

No forms is a user-defined attribute associated with the tray to which this action belongs. **Flag** is a form handling primitive which informs the mailtray owner that the tray requires attention.

Inevitably, the user definition of guards and actions in order to program a message interface incurs a learning overhead. To minimize the effects of this learning overhead an amenable user interface is adopted (see Figure 5). The mailtray systems' user interface utilizes an icon-based representation to allow the easy examination and manipulation of mailtrays. The interface is implemented on a high resolution bit-mapped screen (Sun-3) and uses the direct manipulation of icons via a mouse in addition to keyboard input. A user wishing to interact with our active IPSE would select the appropriate form using the mouse, complete the form as required, and submit the form to the system by *sending* the form via the appropriate tray. Similarly, all responses from the system placed in the appropriate trays by reference to each trays guard definitions.

OBJECT BROWSER AGENT

The IPSE object browser allows users to interact with the system by directly manipulating objects within the knowledge base. The system uses a **dynamic forms** metaphor based on a mapping of objects within the knowledge base to standard format windows (forms) on a workstation screen. It is designed to facilitate a declarative rather than a command-driven style of interaction. To initiate some action, a user supplies information, and the IPSE (specifically the mailer agent) decides what to do with it.

Form-based user interfaces are not a new idea[20], and have been used in data processing systems for many years. Recent research has been devoted to using them as front-ends to systems running on machines equipped with high-definition bit-mapped displays and pointing devices. Cousin-Spice[21] is an example of such a system in which information in a form is structured into two types of field. These embody information representing commands, and that representing parameters. In object-oriented terms, these are analogous to methods and attributes respectively. The former type of field (henceforth referred to as a **button**) is *selected* by the pointing device to execute the command (*pressing* the button), while the latter (henceforth referred to as a **field**) requires a *value* to be associated with it by typing into a space adjacent to the field's label.

In the IPSE object browser, object attributes are represented as fields, and methods as buttons. To assign a value to an object's attribute, the corresponding form field is filled in. To invoke an object's method the corresponding button is pressed.

Dynamic forms are similar to what, in the Information Lens System[22], are called semi-structured messages. It is observed that the structure imposed by the forms, as a framework, can encapsulate much information in fields which would otherwise have to be extracted by the parsing of free text.

Dynamic forms are the physical representation of objects within the knowledge base. Thus, a user filling in a form is actually creating or modifying an object. Any object visible to the knowledge base can be viewed as a dynamic form. Moreover, the powerful concept of constraining attribute relationships enables automatic assistance to be provided to the user. Thus, as a user assigns values to form fields, the system is able to make inferences about the values of associated attributes, both within and across object boundaries. Inferred values may be propagated across attributes as if the form were a spreadsheet.

Dynamic forms are used to create instances of object classes, view existing instances and define new classes. The mechanism is flexible enough to encompass a diversity of object classes, from passive objects, such as files of source code, to classes designed specifically to provide user control of the system, such as forms for sending electronic mail to other users.

Users are not constrained to assign values to every attribute when creating or modifying an object. It is left to the agents which process the object to attempt to make the best sense of information encapsulated by an object. If insufficient information for a particular task is supplied, the IPSE will generate a request for further data.

The mailer agent holds responsibility for the appropriate processing of objects created/modified by the object browser, e.g. compiling an object of class *Ada_source_code* may be performed by filling out a *compile* form, or by pressing the **compile** button on the *Ada_source_code* object's form itself. In either case, the mailer would use its knowledge of object classes and their processing requirements to route the **compile** (<*Ada_source_code* instance>) message to the appropriate (Ada) compiler agent.

The object browser agent (see Figure 6) is composed of a number of components and co-operates closely with the mailer agent, as do all IPSE agents engaged in collaborative problem-solving. The object library component manages co-operation with the mailer agent, and is hence the browser's interface to the IPSE. The user interface

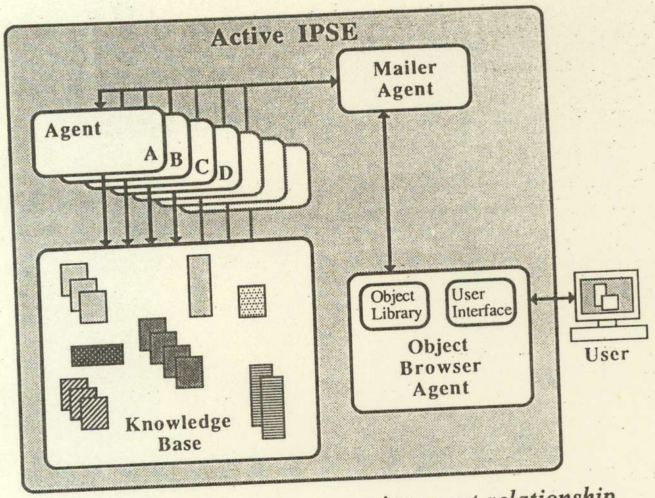


Figure 6. Object browser — environment relationship

component of the object browser is itself defined as a set of objects residing within that portion of the knowledge base for which the object browser is responsible. These objects may be manipulated in the same manner as any other object. Thus, the user's interface to the object browser is reconfigurable and tailorable to individual user's requirements.

The powerful inter-attribute (and inter-object) inferencing mechanism, the IPSE's ability to interpret incomplete data, and the tailorable nature of the user interface, provide the basis of the **Dynamic Forms** interaction metaphor. Dynamic forms are not a unidirectional mechanism for allowing users to provide input to the system. Users and IPSE *interact* via dynamic forms, with the system dynamically interpreting user supplied attribute values and providing run-time assistance by spawning background processes, infering new attribute values, etc.

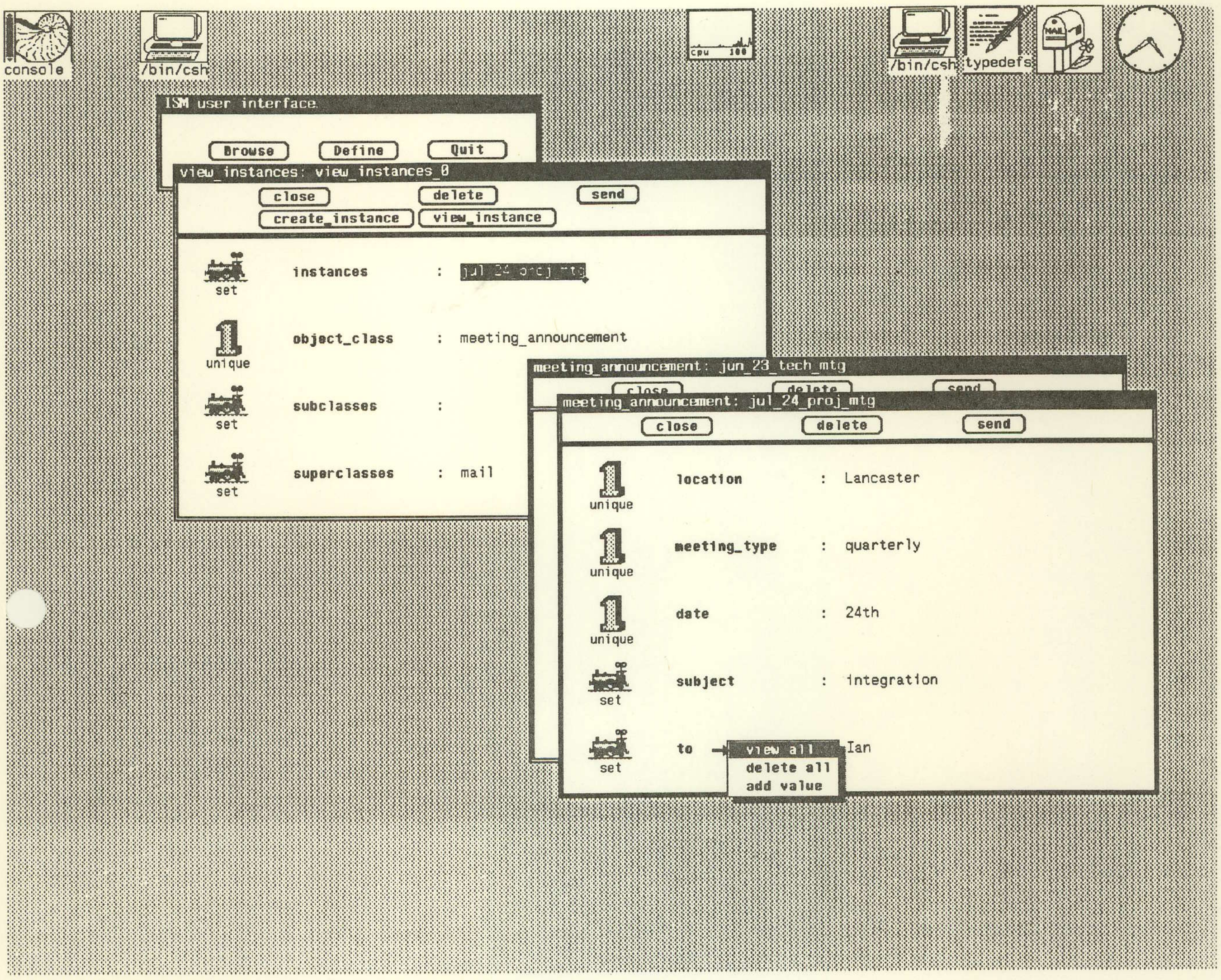
Object browser functionality

The principal requirements of the object browser are that users must be able to inspect the contents of the knowledge base, and manipulate objects within it, i.e. the user needs to be able to find out what is in the object store, look at individual objects, create new objects, modify existing objects (but only if permitted to do so) and define new object classes.

Two components implement the object browser agent:

1. The **user interface**: is responsible for physically presenting objects on the screen, handling user input, mouse clicks, etc. It performs the mapping of objects into forms.
2. An **object library**: is essentially a server to the user interface which forms the object browser's interface to the mailer agent, and hence to the rest of the IPSE. Requests from the user for information about some object or group of objects is relayed by the user interface to the object library. The object library initiates searches of the knowledge base, collects information about the required object(s) and returns the object data to the user.

The mailer agent collaborates with the object library to co-ordinate searching of the knowledge base, and for the delivery of objects to appropriate agents on completion of their manipulation by the users.



```
view_instances: view_instances_0

set      instances      : jul 24 proj mtg
unique   object_class   : meeting_announcement
set      subclasses     :
set      superclasses   : mail

meeting_announcement: jun_23_tech_mtg

meeting_announcement: jul_24_proj_mtg

unique   location       : Lancaster
unique   meeting_type   : quarterly
unique   date           : 24th
set      subject        : integration
set      to             : Ian
```

Figure 7. Viewing an instance of the class meeting__announcement

Browsing example

Browsing the object store may be done on two levels: a definition level, and an instance level. These correspond to looking at what object classes exist, and to viewing instances of a particular class, respectively.

At the instance level users can view individual objects as completed forms, and create new instances by filling in a blank form (Figure 7 illustrates a browsing example). The figure shows the top level ISM control panel, a *view__instance* form which facilitates instance level browsing, and two *meeting__announcement* forms.

The top level ISM control panel is represented by the narrow window labelled **ISM user interface**, and contains the three buttons labelled **Browse, Define** and **Quit:**

- The **browse** button permits objects within the object store to be manipulated, and objects, representing new instances of existing object class definitions, to be created. Browsing in this context encompasses not only the creation and modification of persistent objects such as files of source code and documents, but also the initiation of actions within the system. Initiation of system actions, may take the form of the creation of objects whose sole purpose is to issue instructions to an ISM agent. The creation or modification of a persistent object will also cause the initiation of system actions, however, as this type of operation represents a change to the project state.
- The **define** button permits the definition of new object classes.
- The **quit** button terminates the user's dialogue with the system.

In the scenario represented by Figure 7, the user has pressed the browse button, resulting in the creation of an instance of the *view__instances* class.

The instance of view__instances is represented by the large form at the top left of Figure 7, labelled **Object : view instances__0**, where *view__instances__0* is an automatically generated unique identifier for the instance of *view__instances*.

The form consists of a window containing two subwindows:

1 A control panel containing buttons representing the messages defined for the class view__instances. These are the methods **close, delete** and **send** which are generic to all objects and have the following functionality:
   - The **close** method removes the form from the display.
   - The **delete** method removes the form representing the object from the display and deletes the object

from the knowledge base, releasing the space which is occupied.
- The **send** method removes the form from the display and dispatches the object to the mailer for appropriate processing by other agents.
- The methods **create__instance** and **view__instance** described below.

2 The fields **instances, object__class, subclasses** and **superclasses** corresponding to attributes defined for the class *view__instances*.

An icon adjacent to the attribute name indicates its type. This may be either *unique, set, bag* or *bigtext* (used for unstructured attributes – source code, mail text, etc.).

Attributes of two types are illustrated in the example: unique attributes are denoted by a large figure 1, and set attributes are denoted by a clockwork train (part of a *train* set). By default, the first value of set or bag type attributes is displayed by the form. Users may traverse the list of values by clicking a mouse button on the attribute name. Alternatively, a full listing of the values may be viewed by selecting the *view all* option from a menu defined for set and bag attributes. Similar facilities exist to permit the viewing of bigtext attributes.

Viewing an instance

In the example, a user has assigned a value to the **class** attribute, indicating that he wishes to inspect instances of the class *meeting__announcement*.

A constraint associated with the **class** attribute enables values of the other attributes to be automatically inferred. The following values have been generated:

- The attribute **superclasses** has been assigned a set of values including *mail*, which represents the most immediate superclass.
- A *null* value has been generated for the **subclasses** attribute indicating that the *meeting__announcement* class represents a leaf of its inheritance tree.
- A full set of values for the **instances** attribute have been generated. These represent instances of the class *meeting__announcement* resident within the knowledge base.

In the example, the user has opted to inspect two meeting__announcement objects. Pressing the view__instance button causes the object currently listed by the instances attribute to be displayed. The object library responds by searching the object store for the required object and returning its details to the object browser. The object browser maps these onto the corresponding form representation.

The meeting announcements requested by the user are represented by the two forms labelled **meeting announcement: jun__23__tech__mtg** and **meeting__announcement: jul__24__proj__mtg** at the bottom right of the figure.

The **to** attribute belonging to the jul__24__proj__mtg form illustrates the menu options associated with set and bag attributes. The options available are **view all, delete all** and **add value**. These supplement the editing operations which may be directly applied to a selected value, namely to delete or modify it.

Whether such editing operations on values may be accepted by the browser is dependent on the constraints associated with the attribute definitions. Some objects (e.g. those representing archive components of a project's milestones) may be immutable and have protected attribute values. Such a case would spawn a warning message object, displayed (again, using our dynamic forms metaphor) as a form. For other classes of object it may be appropriate to allow objects to be modified.

Figure 7 illustrates the use of the *view__instances* object class for viewing existing objects. The *view__instances* form may also be used to create new instances of a class.

The **create__instance** method causes a blank form representing the class *meeting__announcement* to be displayed. An additional field, **instance__name**, is displayed to which the user may assign a unique object identifier. Failure to do so will result in ISM automatically generating a unique object identifier.

On completion of the form the **send** button is pressed. The object library creates the object (mapping the form contents onto the class definition), which is then dispatched to the mailer agent for processing.

CONCLUSION

The design of the next generation of IPSEs will differ greatly to that of previous generations. The evolutionary design which has led to the development of current IPSEs from programming environments is not capable of accommodating the emerging technologies which simplify automation of the whole software process.

The prototype IPSE described in this paper explores the potential for exploiting knowledge based programming techniques for the attainment of a high degree of integration within the IPSE. It is postulated that a federation of intelligent, co-operating agents form an appropriate architecture for the IPSEs of the near future. The IPSE has been designed using an open systems philosophy, such that new intelligent agents can be added incrementally as technology enables their production. Thus, tasks which may currently require a human to perform them may be devolved to computer agents in the future.

The key to achieving a sufficiently high degree of integration and co-operation within the IPSE is the use of a consistent communication mechanism among agents and users. An intelligent mailer agent has been designed to enable the kind of group communications which must take place in such an environment. A tailorable user interface to the communication facilities permits users to interact with the system in such a way that its internal details are made transparent.

ACKNOWLEDGEMENTS

Thanks are due to our collaborators in the ISM project, Software Sciences Ltd and the University of Keele. The research work is funded by the Alvey Directorate, UK.

REFERENCES

1 **Bourne, S R** *The Unix System* Addison-Wesley, USA (1982)

2 **Hall, J A, Hitchcock, P and Took, R** 'An overview of the ASPECT architecture' in **McDermid, J (ed)** *Integrated Project Support Environments* IEE software engineers series 1 (1985)

3 **Alderson, A, Bott, M F and Falla, M E** 'An overview of the ECLIPSE project' in **McDermid, J (ed)** *Inte-*

grated Project Support Environments IEE software engineering series 1 (1985)

Winograd, T 'Breaking the complexity barrier (again)' *Proc. ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages — Information Retrieval* Maryland, USA (1973)

The ISM Project Consortium 'The ISM project: towards a knowledge-based IPSE' The KBS Group, Software Sciences Ltd. (1987)

Hewitt, C 'Viewing control structures as patterns of passing messages' *Artif. Intell.* Vol 8 No 3 (1977)

Malone, T W, Grant, K R Turbak, F A, Brobst, S A and Cohen, M D 'Intelligent information sharing systems' *Commun. ACM* Vol 30 No 5 (1987)

Schneiderman, B 'The future of interactive systems and the emergence of direct manipulation' *Behaviour & Info. Technol.* Vol 1 No 3 (1982)

Rentsch, T 'Object oriented programming' *ACM SIGPLAN Notices* OOPS 80 (1980)

Wegner, P 'Dimensions of object-based language design' *Proc. OOPSLA'87* ACM press, USA (1987)

Minsky, M 'A framework for representing knowledge' in **Winston, P (ed)** *The Psychology of Computer Vi[illegible]* McGraw-Hill, USA (1975)

2 **Ha[illegible]on, S J, Kraut, R E and Farber, J M** 'Interface design and multivariate analysis of Unix command use' *ACM Trans. Office Automation Syst.* Vol 2 No 1 (1984)

3 **Reid, P and Welland, R C** 'Project development in view' in **Sommerville, I (ed)** *Software Engineering Environments* Peter Perigrinus, UK (1986)

4 **Greif, I and Sarin, S** 'Data sharing in group work' ACM *Trans. Office Info. Syst.* Vol 5 No 2 (April 1987) pp 187–211

15 **Hiltz, S R and Turoff, M** 'Structuring computer-mediated communication systems to avoid information overload' *Commun. ACM* Vol 28 No 7 (July 1985)

16 **Hutchinson, D, Armitage, R and Muir, S J** 'A user agent for the Unix mail system' *Data Processing* Vol 28 No 10 (December 1986)

17 **Brobst, S A, Malone, T W, Grant, K and Cohen, D** 'Toward intelligent message routing systems' in **Uhlig, R (ed)** *Computer Message Systems 85 — Proc. 2nd Int. Symposium on Comput. Message Syst.* North-Holland, The Netherlands (1986)

18 **Sumner, M** 'A workstation case study' *Datamation* (February 15 1986) pp 71–79

19 **Rodden, T and Sommerville, I** 'Mailtrays: an object orientated approach to message handling' *European Conf. Info. Technol. for Organizational Syst.* Athens, Greece (May 16–20 1988)

20 **Smith, C D, Irby, C, Kimball, R, Verplank, W and Harslem, E** 'Designing the star user interface' in **Degan, P and Sandwall, E (eds)** *Integrated Interactive Computing Systems*, North-Holland, The Netherlands (1983)

21 **Hayes, P J and Szekely, P A** 'Graceful interaction through the COUSIN command interface' *Int. J. Man-Machine Studies* Vol 19 (1983)

22 **Malone, T W, Grant, K R and Turbak, F A** 'The information lens: an intelligent system for information sharing in organisations' *Proc. CHI'86* (1986)

Expert

Ovum Ltd.,

This report
leading bank
ing artificia
tions. Som
already invo
are watchin
interest. Th
systems to b
base techn
reasons for t

- Banking a
  people b
  informati
  importan
  houses n
  of their b
- These i
  computer
  take adv
  and comp
- Increasin
  national
  tasks ne
  support,
  help to fa
- Banking
  ing more
  improven
  service is

Market val

A survey by
the banking
found that
active in b
and nearly
approach. I

Expert

Ovum Ltd,

Expert syste
as one of
nologies of
growing mu
ware busines
also conclu
becoming
national,
suppliers st
are likely to
The fifth