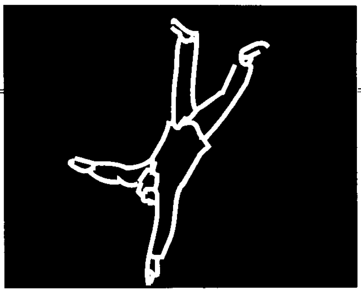


Process Integration in CASE Environments

PEIWEI MI and WALT SCACCHI, University of Southern California

◆*At a higher level than tool or object integration, process integration makes the implicit chain of development tasks explicit. This lets us develop process-driven software environments.*

Research in CASE environments has focused on two kinds of integration: tool and object. Tool integration deals with the implicit invocation and control of development tools.[1-3] Object integration seeks to provide a consistent view of development artifacts and easy-to-use interfaces to generate, access, and control them.[4-5]

We propose a higher level of integration, *process integration*, which represents development activities explicitly in a *software process* model to guide and coordinate development and to integrate tools and objects. A CASE environment based on process integration is called a *process-driven* CASE environment. Our implementation strategy is to realize process integration using existing CASE environments or tools.

PROCESS INTEGRATION

Today's CASE environments generally support some form of tool or object integration. Tool integration provides a development tool set and an invocation mechanism that controls its use. In fact, the tools form a *conceptual tool-invocation chain* that facilitates related development activities. In Unix environments, for example, programming is supported by text editors, language compilers, object linkers, program debuggers, and shell programs.

The invocation chain is conceptual because it does not have an explicit representation in most CASE environments. Furthermore, every programmer has their own version of an invocation chain that has emerged through personal experience. However, for a small tool set, invocation chains vary only slightly.

Object integration is based on an object model of software artifacts and emphasizes artifact management. The production and consumption of these artifacts normally has a partial order — an artifact

0740-7459/92/0300/0045/$03.00 © IEEE

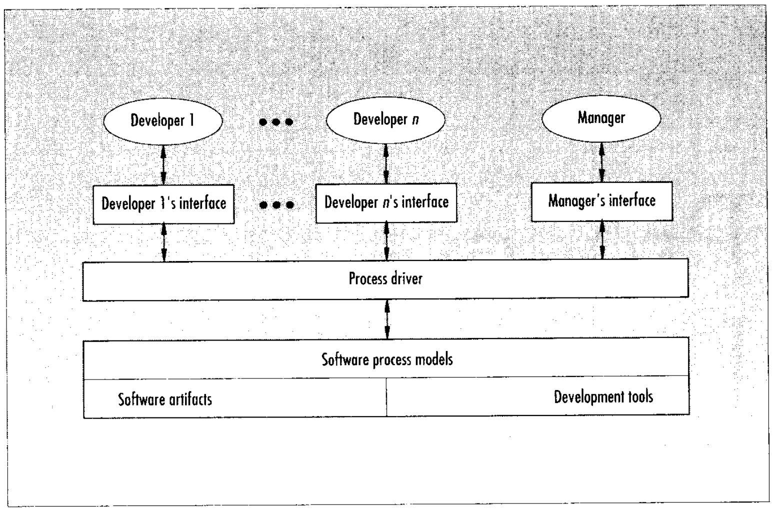


Figure 1. *Architecture of a process-driven CASE environment.*

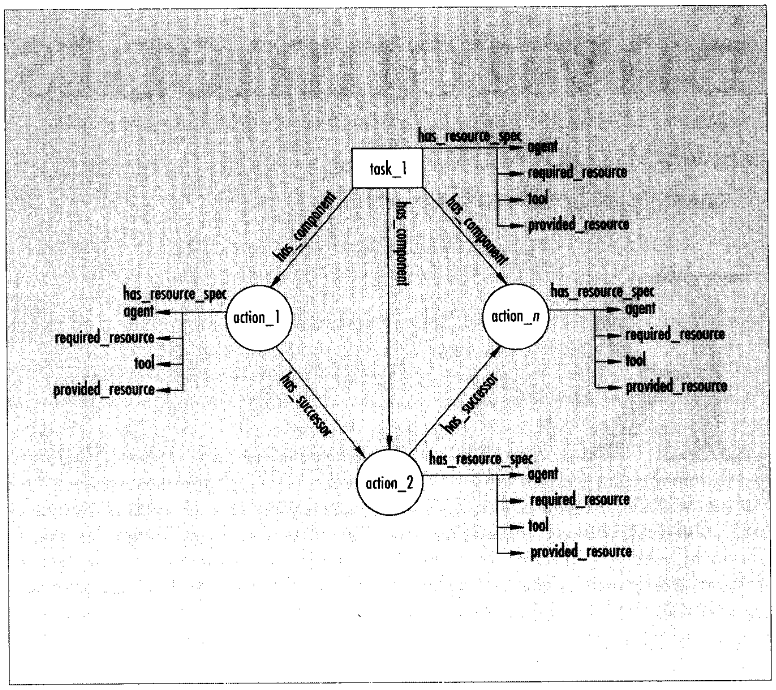


Figure 2. *A sample software process model, which is a repository of information on the status of processes and activities. An SPM is a hierarchy of tasks, subtasks, and actions. The top level is a task, depicted as a rectangle, that is recursively decomposed into subtasks. At the bottom of the hierarchy are actions, depicted as a circle, which are either single tool invocations or simple resource transformations. Each level specifies a partial order for subtask execution. An SPM also indicates four categories of resource requirements for a subtask, which are represented as independent object classes with relations that link them to the SPM.*

produced early in the life cycle will be used later to create another artifact. For example, an early artifact in system development often is the informal requirements description. Intermediate artifacts are the requirements specification and the architectural design. The final artifact is the system itself.

In this case there is a *conceptual resource-transformation chain* that progresses from initial artifacts to intermediate ones and then to the final product. On the other hand, object integration does not address the activities that produce and consume artifacts.

A common feature of these two conceptual chains is that they describe *task execution*, albeit from different perspectives. The tool-invocation chain represents the use of tools in task execution, while the resource-transformation chain represents the task-execution I/O.

Process integration seeks to make the conceptual task-execution chain explicit, flexible, and reusable. We represent the task-execution chain as a software process model. By using an SPM representation, we can achieve more integrated CASE environments.

Process integration provides mechanisms to guide the software process and manage workspaces, tool invocations, and objects. Process integration also lets software managers monitor and control the progress of development. As Figure 1 shows, the key mechanisms for process integration are SPMs, a process driver, and interfaces for both the developer and manager. These are the key components of the architecture for a process-driven CASE environment.

SOFTWARE PROCESS MODEL

An SPM is a formal way to organize and describe how life-cycle models, development methods, software artifacts, CASE tools, and developers fit together — it is a collection of objects representing activities, artifacts, tools, and developers. Each SPM object has its own representation and describes a kind of information that is involved in software development. SPM objects are linked by many kinds of relations. Figure 2 shows the structure of a sample SPM.

An SPM, which we have described in detail elsewhere,[6] is a repository of information on the status of the processes and activities manipulated throughout a development project. It specifies an *activity hierarchy* and *resource requirements*.

An activity hierarchy decomposes an SPM into a hierarchy of smaller activities, *tasks* and *actions*. How many decomposition levels a project has is arbitrary and depends on the project's complexity. The top level of an SPM is a task that is recursively decomposed into a set of interrelated subtasks (which include tasks and actions). At the bottom of the hierarchy are actions, which are either single tool invocations or simple resource transformations.

Each level specifies a partial order for subtask execution. The execution order defines several types of precedence relationships among subtasks, including sequential, parallel, iterative, and conditional.

An SPM also indicates four categories of resource requirements for a subtask to be performed. As Figure 2 illustrates, an SPM models the

- ♦ various developer and organizational roles users take while performing a subtask,
- ♦ software artifacts that are needed (*required* resources) and created or enhanced (*provided* resources) during a subtask,
- ♦ tools used, and
- ♦ information about a subtask's schedule and its expected duration.

These resource requirements are represented as independent object classes with relations that link them to the SPM. For example, a system's product model might be defined to have a module-decomposition structure, with its modules linked to their producer and consumer subtasks.

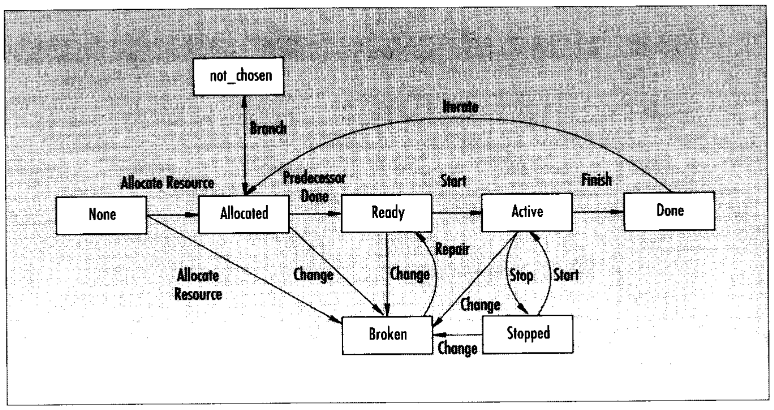


Figure 3. A status-transition graph, which is the internal representation of tasks, subtasks and actions. Boxes denote status values (defined in Table 1), arcs denote transitions that change those values and the result, and arc labels denote the transition type. The process driver implements the status-transition graph, and each status-transition graph is in turn represented as a node in the higher level SPM.

PROCESS DRIVER

A process driver interprets and executes an SPM according to its activity hierarchy. It manages the order of subtasks and the constraints to be satisfied before they can be performed. It plays the role of an automated driver in that it initiates an SPM, enacts its subtasks, accepts developers' inputs, updates and propagates the status of these subtasks, and triggers other subtasks.

A key variable that represents the state of process enactment is *status*, which is attached to each task or action in an SPM. The value of an SPM's or subtask's status indicates its current development state. The status variable is updated by the process driver on the basis of interaction with developers. Updates to a subtask's current status value represent the subtask's progress through enactment. To this end, process execution drives the status of an SPM from None to Done.

Table 1 lists process status values and their definitions for both actions and tasks. An action's status is primitive and is determined by operations on it; a task's status is recursively defined and is determined by the status of its component subtasks. The process driver updates these values according to input from an SPM and developers. Thus, while an SPM indicates a prescribed order of subtask enactment, the record of status transitions describes the order in which enactment actually occurred.

Figure 3 shows status transitions initiated by the process driver. In Figure 3, the

TABLE 1
STATUS VALUES AND DEFINITIONS

Value	Definition for actions	Definition for tasks
None	Initially set	Subtasks have status None
Allocated	Developers, tools, and required resources have been allocated	Subtasks have status Allocated
Ready	Allocated, either without predecessors or its precedecessors are done	Some subtasks are Ready, but none are Active
Active	Enactment is in progress; being performed by assigned developers.	Some subtasks are Active
Stopped	Not being performed; voluntarily stopped	Some subtasks are Stopped, but none are Active, Broken, or Ready
Broken	Either one or more required resources is unavailable or it is unable to continue for some reason	Some subtasks are Broken, but none are Active or Ready
Done	Execution has finished successfully	Subtasks have status Done
Not chosen	Not selected for execution	Not available

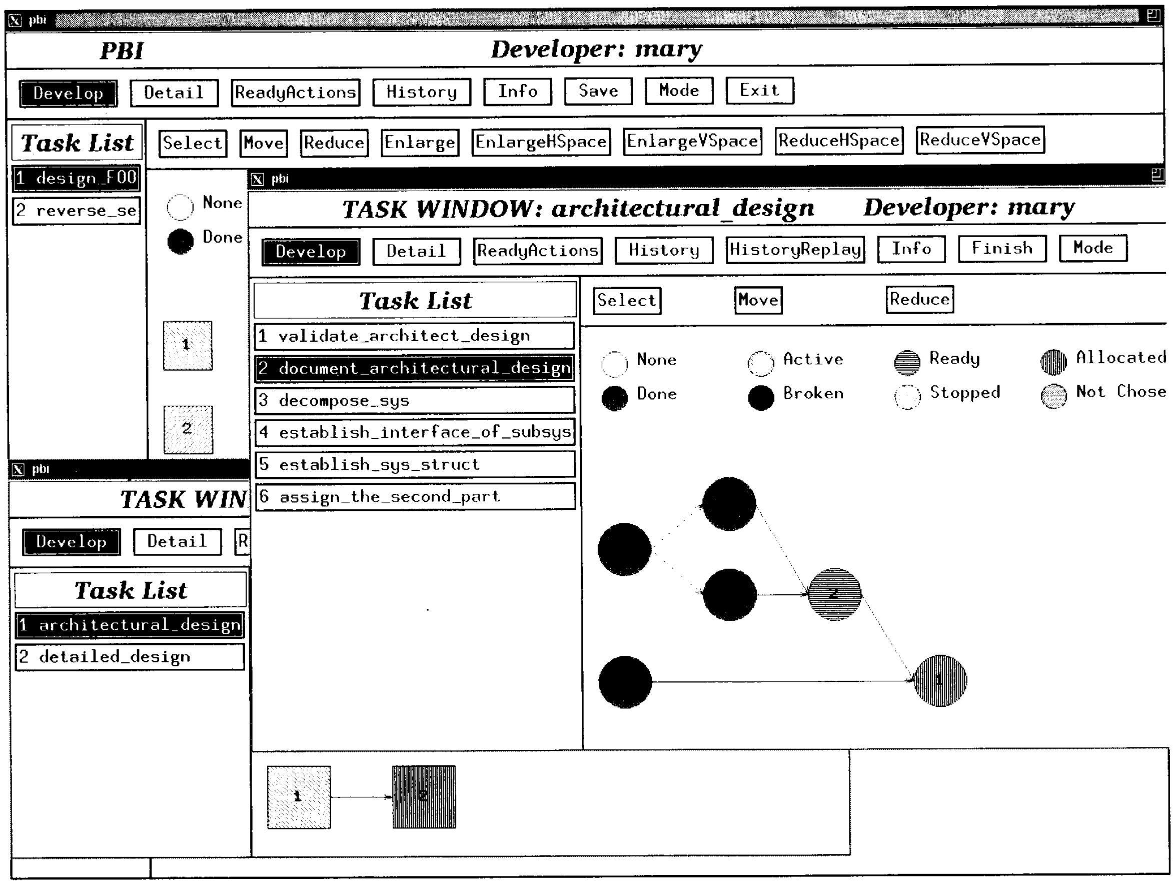


Figure 4. A stack of task windows in the developer's interface. The graph in the top window is an SPM showing actions and their status.

boxes denote status values and the arcs connecting them denote enactment transitions that change those values and the result. Arcs are labeled with transition types. For example, action Start changes an action's status from either Ready or Stopped to Active, signaling that the action is being executed. The process driver implements the status-transition graph. Each status-transition graph is in turn represented as a node in the higher level SPM.

Once managers have allocated resources to a project, developers start an SPM's enactment by informing the process driver, which issues a Start signal on initial Ready actions. The action's status is thereby changed to Active. As actions are performed, some of them may be Done, others may be Stopped, and still others will be Broken, which means they need repairing or rescheduling actions to recover.[7] When actions are Done, they trigger the process driver to update the SPM and assert more Ready actions. This enactment continues until the entire SPM is Done. During enactment, managers monitor progress by observing the status change. Managers can elect to change the enacted SPM or its resource allocation at any time. Accordingly, developers enact and managers control SPMs through two separate user interfaces.

DEVELOPER'S INTERFACE

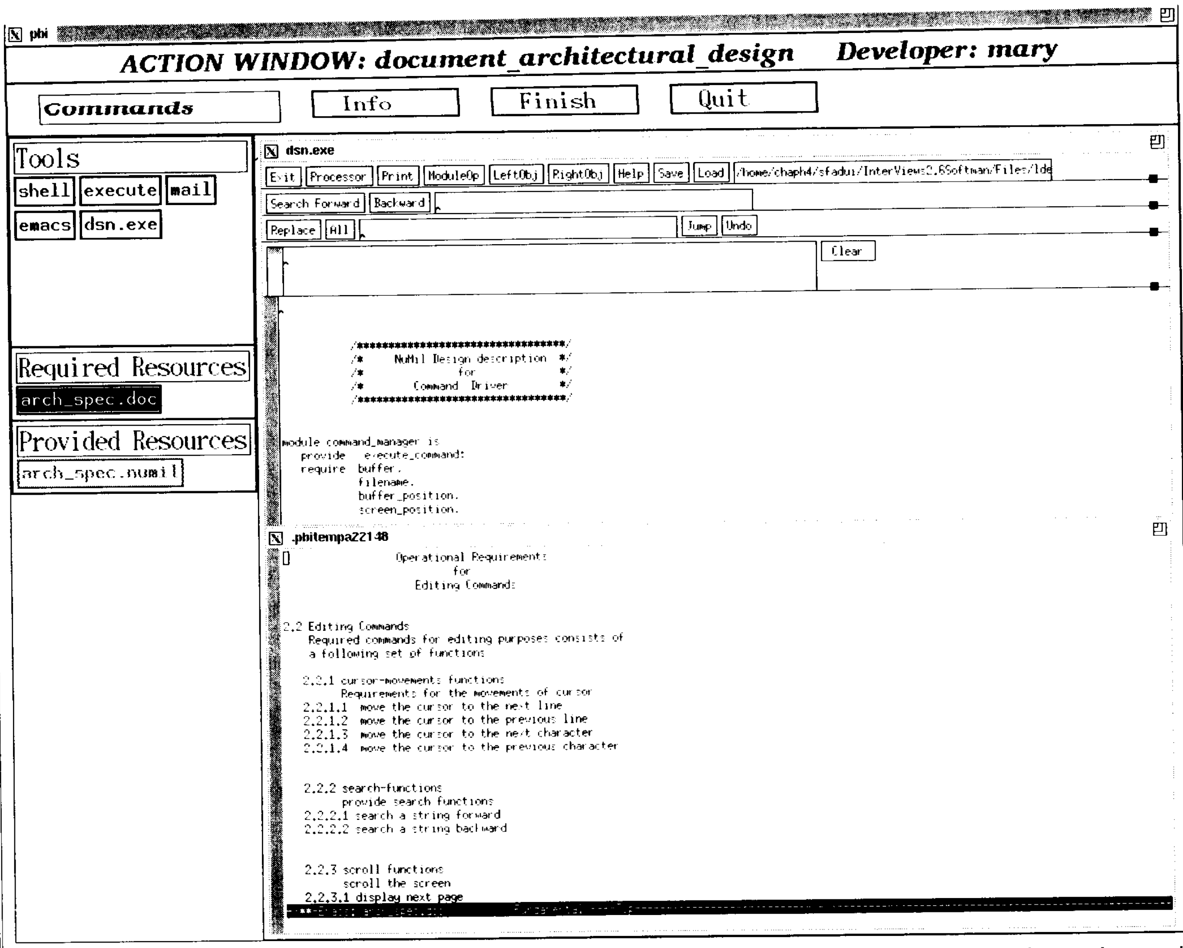
The developer's interface is a working environment that lets developers enact an SPM. Different developers execute SPMs concurrently and perform only their assigned subtasks through process guidance and workspaces.

Process guidance. Process guidance informs developers of the subtasks they can perform and when these subtasks are ready to start. The developer's interface supports process guidance through navigation within an SPM by the process driver.

A developer can start a subtask only if its status is Ready or Stopped. When a subtask is finished, the developer's interface sends a Done signal to the process driver, which uses the signal to determine if successor Allocated subtasks are Ready. This lets developers start assigned tasks as soon as they become Ready.

The developer's interface provides process guidance through task windows, which show both explicit and implicit process representation. A task window presents a task's subtasks and their current status. Figure 4 shows a stack of task windows with an example of an SPM with explicit process representation.

Each task window displays the task's immediate subtasks as an SPM: Tasks are represented as squares, actions as circles, and precedence relationships as arrows from a subtask to its successor subtasks. The status of a subtask is indicated with different colors on a color display and with different icons on a monochrome display. Therefore, an SPM indicates both task-execution order and how enactment is progressing.



```
/***********************************/
/*     NuMil Design description    */
/*               for               */
/*          Command  Driver        */
/***********************************/


module command_manager is
   provide   execute_command:
   require  buffer.
            filename.
            buffer_position.
            screen_position.
```

```
                 Operational Requirements
                           for
                     Editing Commands


2.2 Editing Commands
    Required commands for editing purposes consists of
    a following set of functions

  2.2.1 cursor-movements functions
        Requirements for the movements of cursor
  2.2.1.1  move the cursor to the next line
  2.2.1.2  move the cursor to the previous line
  2.2.1.3  move the cursor to the next character
  2.2.1.4  move the cursor to the previous character


  2.2.2 search-functions
        provide search functions
  2.2.2.1 search a string forward
  2.2.2.2 search a string backward


  2.2.3 scroll functions
        scroll the screen
  2.2.3.1 display next page
```

***Figure 5.** An action window in the developer's interface. The action is document_architectural_design. The developer has invoked two tools — dsn.exe and emacs, each operating on one resource — arch_spec.numil and arch_spec.doc, respectively.*

A developer can start the execution of any Ready subtask by clicking on it, which opens another window. If the selected subtask is a task, the new window is a task window that shows a lower decomposition level. For example, Mary's developer interface in Figure 4 shows, in the task-list window in the upper left, the two SPMs assigned to her: design_FOO and reverse_se.

Mary has opened two levels of decomposition, one for design_FOO in the lower left window and one for architectural_design in the center window. In the center window, the architectural_design subtask is decomposed into a list of six actions, whose icons indicate that four are Done, one is Ready, and one is Allocated.

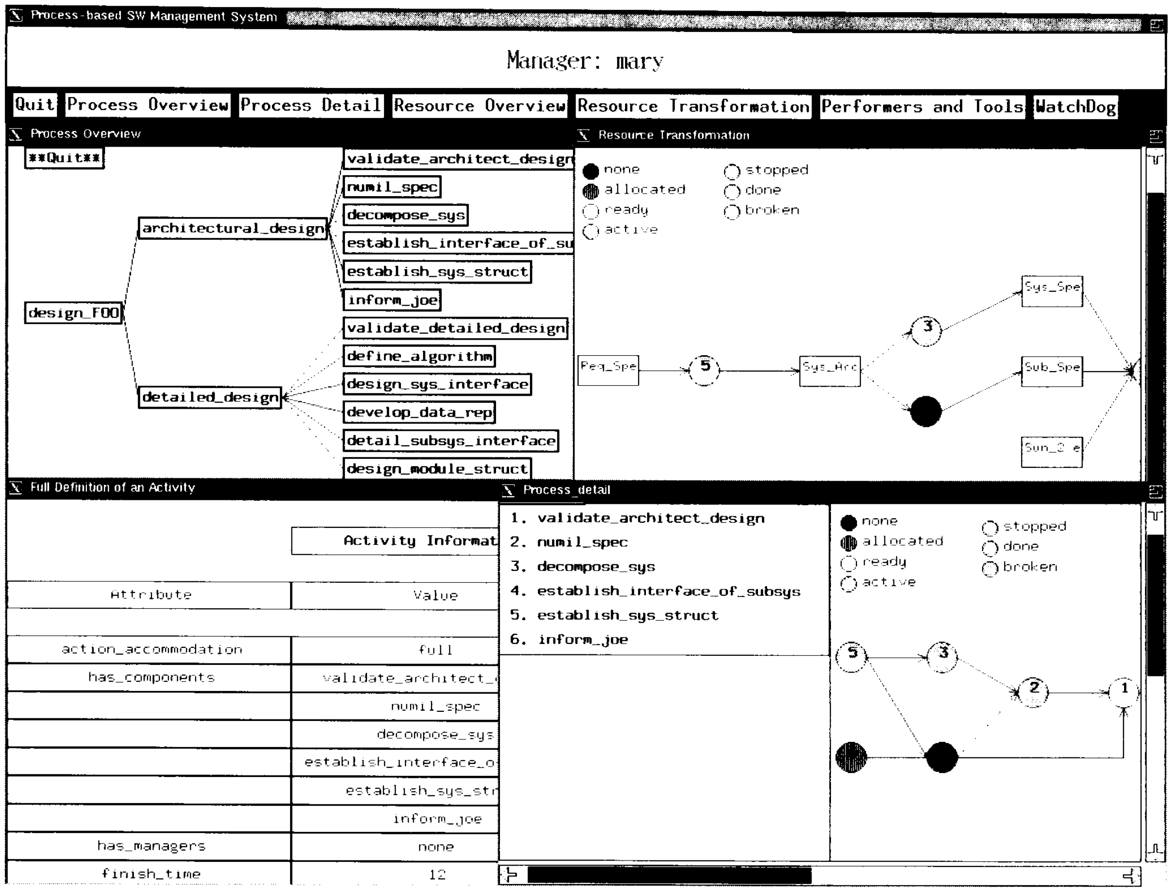
On the other hand, if the developer selects an action subtask, the new window is an action window, which provides a working environment.

A task window that shows an SPM with an implicit process representation would display a list of Ready actions, called a ready list, and an action window. The developer selects an action from the ready list and finishes it in the action window's workspace. The developer's interface updates the Ready list when an action is Done so the next Ready actions can be displayed. This implict process representation is a shortcut for developers who prefer not to be guided by an explicit process representation. The developer's interface nonetheless enforces the process description implicitly.

Workspaces. In the developer's interface, a workspace is used to perform an action and is accessed through an action window, as Figure 5 shows. Creating a workspace has two aspects: setting up a work area and invoking the necessary tools with resources as parameters.

Selecting an action from an SPM causes its required resources, its provided resources, and its associated tools to be displayed in an action window. An action window also lets developers invoke the tools on specified resources. Furthermore, when an action specifies only one tool and one resource, tool invocation is automatic: The developer's interface invokes the tool when the action window opens.

Figure 5 shows the action window for the action document_architectural_design. In the window, two tools — dsn.exe and emacs — have been invoked. Each tool operates on one resource — arch_spec.numil and arch_spec.doc, respectively. When an action is finished, the action window gets its provided resources from the tools and puts them into the correct position according to the specification in the SPM so they can be used in successor subtasks.



Activity Informat

Attribute	Value
action_accommodation	full
has_components	validate_architect_
	numil_spec
	decompose_sys
	establish_interface_o
	establish_sys_str
	inform_joe
has_managers	none
finish_time	12

1. validate_architect_design
2. numil_spec
3. decompose_sys
4. establish_interface_of_subsys
5. establish_sys_struct
6. inform_joe

Figure 6. Manager's interface, showing (top left) an activity graph for design_FOO; (top right) a graph of resource production and consumption; (bottom right) a task-predecence graph for a group of subtasks and their current status; and (bottom left) the interface where the manager can modify the values of attributes in an SPM to assign tasks and allocate resources.

MANAGER'S INTERFACE

The manager's interface gives managers and analysts the tools to define, monitor, and control the SPMs that developers are working on concurrently.

From the process driver's point of view, a manager's interface does not change the status of an SPM. Instead, it initializes an SPM to ensure that the minimum required resources needed to start it are allocated and to monitor the progress of a process. Process managers use a different interface (not shown) to create, prototype, analyze, and simulate SPMs.[7]

The manager's interface simply retrieves information from the SPM and presents it in easy-to-understand graphs and tables, as Figure 6 shows. The process driver can update these graphs and tables in real time, as developers work.

Three windows in Figure 6 are concerned with monitoring the progress of things like subtask completion and resource creation. The upper left window shows an activity graph for design_FOO; the lower right window is a graph of the task-precedence relationships among a group of subtasks and their current status; the upper right window shows the resource production and consumption relationships among a task's subtasks.

The lower left window in Figure 6 is concerned with controlling the process, which involves changing the values of an SPM's attributes. This modification function lets a manager assign tasks and allocate resources.

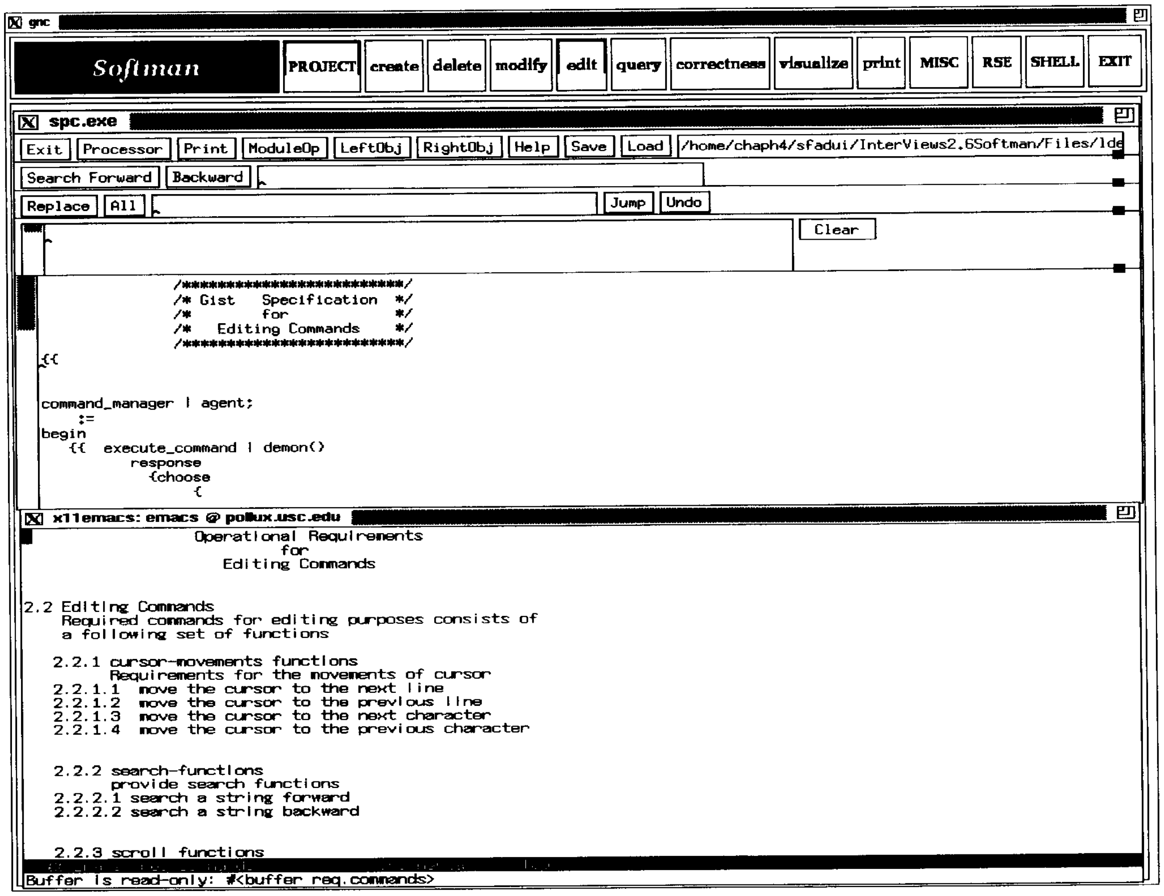
SOFTMAN EXPERIMENT

An example of our strategy to implement process-driven CASE environments with existing CASE environments[8] is an experiment involving the Softman environment,[5] which was developed as part of the University of Southern California's System Factory project (although other CASE environments or tools could have served as well).

Softman is an integrated CASE environment for forward and reverse engineering large software systems. Its comprehensive set of support mechanisms and tools makes it a powerful environment for large-scale development. However, its development methodology and its tools can be difficult to learn. A process driven Softman environment can overcome, or mitigate, these difficulties.

We made Softman process driven by

1. Identifying its basic concepts, functions, component tools, and tool-invocation sequences.
2. Formally representing the information gathered in Step 1 in an SPM called the Softman process model.
3. Porting the Softman process model to a process-driven CASE environment



```
                /***************************/
                /* Gist   Specification   */
                /*          for            */
                /*   Editing Commands      */
                /***************************/
{{

command_manager | agent;
    :=
begin
   {{  execute_command | demon()
          response
            {choose
                 {
```

```
                    Operational Requirements
                              for
                        Editing Commands


2.2 Editing Commands
    Required commands for editing purposes consists of
    a following set of functions

   2.2.1 cursor-movements functions
         Requirements for the movements of cursor
   2.2.1.1  move the cursor to the next line
   2.2.1.2  move the cursor to the previous line
   2.2.1.3  move the cursor to the next character
   2.2.1.4  move the cursor to the previous character


   2.2.2 search-functions
         provide search functions
   2.2.2.1 search a string forward
   2.2.2.2 search a string backward


   2.2.3 scroll functions
Buffer is read-only: #<buffer req.commands>
```

Figure 7. *Softman's user interface. The top window shows the editing of a design specification; the bottom window is a view of operational requirements.*

and integrating Softman's object model and tool set.

Identifying concepts. Figure 7 shows Softman's user interface. The top window shows the editing of a design specification; the bottom window is a view of operational requirements.

Through extensive studies, we identified these concepts as important to Softman:

♦ Softman supports an incremental, iterative tool-based methodology. It creates and manipulates a tree-structured collection of life-cycle documents such as requirements analyses, requirements specifications, designs, implementations, tests, and maintenance information. In Softman, development stages are decomposed until you identify the level of a single tool invocation.

♦ Softman's product model is a collection of life-cycle objects and their attributes and relationships.[9] It also provides more detailed information so that all objects have attributes that characterize their interfaces and interconnections. Softman requires that you create an object's skeletal structure, called mod_struct, before you can create and maintain its content, called mod_cont, correctly. We call this a structure-first, content-second development method.

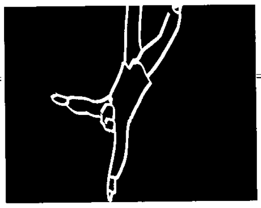
♦ For each development stage, Softman provides a set of structure-oriented, correctness-checking tools. You use these tools to create, modify, and maintain both an object's structure and content.[5] Softman's tools include language-directed editors to create object content and graphical editors to manipulate object structures. These tools communicate with an object-management repository that checks consistency, completeness, and traceability constraints on all objects so as to incrementally assure or track their (in)correctness.

This first step is the most critical. Here, the process analyst must understand the modeled process. The analyst must talk with people who have extensive knowledge of the process that the environment will support. We worked with the original Softman developers.

Formal representation. We defined the Softman process model to include development activities, their prescribed execution order, their required tools (including multiple tool choices when necessary), and their associated required and provided objects. Overall, the Softman process model follows Softman's structure-first, content-second methodology.

Figure 8 gives the specification of an action, des_mod_cont, within the Softman process model. The action definition specifies its developer, execution order, required object, provided object, and two development tools.

This step is relatively easy, given the knowledge acquired in step 1. We are now building more tools to help automate this step.



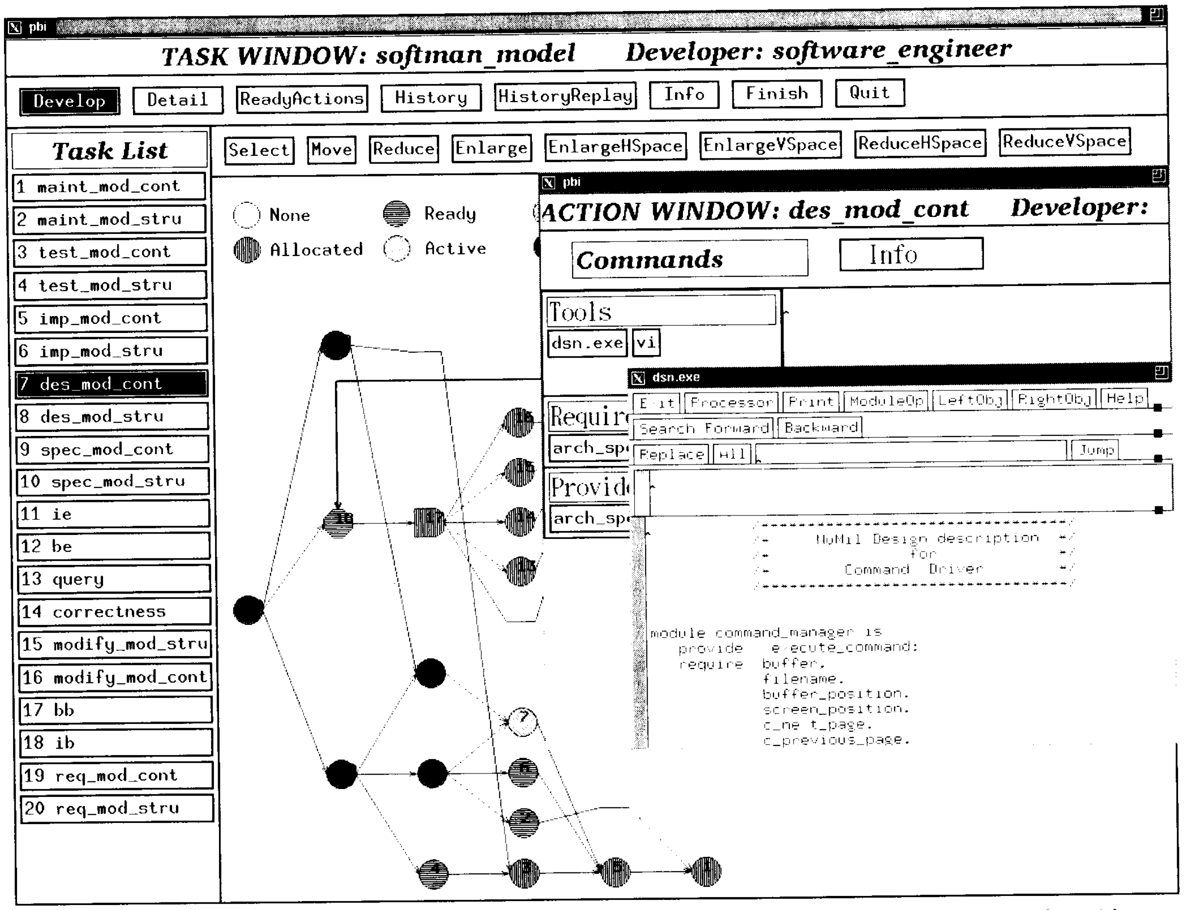
```
{{ des_mod_cont
    INSTANCE: ACTION
    TASK_ASSIGNED_TO_AGENT_ROLE: software_engineer
    TASK_COMPONENT_OF: softman_model
    TASK_HAS_PREDECESSOR: des_mod_struct
    TASK_HAS_SUCCESSOR: imp_mod_cont
    TASK_REQUIRE_RESOURCE: arch_spec.doc
    TASK_PROVIDE_RESOURCE: arch_spec.numil
    TASK_REQUIRE_TOOL: vi dsn.exe}}
```

Figure 8. The specification of an action, des_mod_cont, within the Softman process model. The action definition specifies its developer, execution order, required object, provided object, and two development tools.

Porting the process model. Porting the Softman process model into a process-driven CASE environment is straightforward because the model is simply an instance of a defined SPM class.

In this step, we input the Softman process model into the process driver and made the Softman data model known to the driver. We then reused and integrated Softman's tool set by reattaching the Softman tool-invocation menu items to the Softman developer's interface and establishing links between the tools and the data model.

The problems that emerge in this step are how to invoke the tools and how to reconcile potential tool incompatibilities.



```
/*-----------------------------------*/
/*     NuMil Design description     */
/*              for                 */
/*        Command  Driver           */
/*-----------------------------------*/


module command_manager is
   provide   e ecute_command:
   require  buffer.
            filename.
            buffer_position.
            screen_position.
            c_ne t_page.
            c_previous_page.
```

Figure 9. Developer's interface to the process-driven Softman environment. The bottom window shows the Softman process model; the action window at right executes des_mod_cont, defined in Figure 8.

We examine some of these tool-integration problems later.

Figure 9 shows the developer's interface to the new process-driven Softman environment. The bottom window shows the Softman process model; the action window at right executes des_mod_cont, defined in Figure 8.

The new Softman environment uses the interfaces and supports the Softman process model as well as reuses the original Softman data model and tool set. The process-driven Softman environment preserves the important Softman concepts and functions and adds support for process guidance and project management.

When they use the process-driven Softman environment, developers know what development stage they are performing, the tools available to them, the software documents they must produce, and the task they are to perform next.

We have delivered the new process-driven Softman environment to some industrial organizations and are now experimenting with it. We are also incorporating support for integrating distributed object repositories and heterogeneous data models.[10]

Observations. During our experiment, we observed that the process-driven Softman environment is flexible to change. In fact, we have defined variations of the Softman process model to support different development methodologies but use the same product model and tool set. This would not be possible with the original Softman CASE environment without a major reprogramming effort.

We also observed that this strategy for process integration should be applicable to other CASE environments that provide basic tool- and object-integration mechanisms.

Finally, we observed that the interfaces of existing development tools must be highly compatible for them to appear to be seamlessly integrated in a process-driven CASE environment. Our current strategy is to enable each CASE environment to define its own product and tool models and to have an open structure for process integration to incorporate them. The Softman experiment has shown promise in this regard, but more study is needed.

Process integration uses a few key components to form the backbone of a process-driven CASE environment. The key components are SPMs, a process driver, a tool set, and interfaces for both developers and managers.

Process integration supports an open system structure that we believe can be added to other CASE environments with reasonable effort.

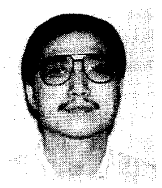
Our successful migration of the Softman environment to a process-driven CASE environment proves the feasibility of our strategy. ♦

ACKNOWLEDGMENTS

This work was supported in part by contracts and grants from AT&T Bell Laboratories, Northrop, Pacific Bell, and the Office of Naval Technology through the Naval Ocean Systems Center. Anthony Karrer, Thomas Getzinger, Mary Li, Junhui Luo, and Yu-ling Young contributed to the implementation effort.

REFERENCES

1. G. Boudier et al., "An Overview of PCTE and PCTE+," *SIGPlan Notices*, Feb. 1989, pp. 226-227.
2. D. Garlan and E. Ilias, "Low-Cost, Adaptable Tool Integration Policies for Integrated Environments," *Proc. SIGSoft Symp. Software Development Environments*, ACM Press, New York, 1990, pp. 1-10.
3. I. Thomas, "Tool Integration in the PACT Environment," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 13-22.
4. E.W. Adams, M. Honda, and T.C. Miller, "Object Management in a CASE Environment," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 154-165.
5. S.C. Choi and W. Scacchi, "Softman: An Environment for Forward and Reverse CASE," *Information and Software Technology*, Nov. 1991, pp. 664-674.
6. P. Mi and W. Scacchi, "A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes," *IEEE Trans. Knowledge and Data Eng.*, Sept. 1990, pp. 283-294.
7. P. Mi and W. Scacchi, "Modeling Articulation Work in Software Engineering Processes," *Proc. Int'l Conf. Software Process*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 188-201.
8. C. Fernstrom and L. Ohlsson, "Integration Needs in Process-Enacted Environments," *Proc. Int'l Conf. Software Process*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 142-158.
9. S.C. Choi and W. Scacchi, "Assuring the Correctness of Configured Software Descriptions," *ACM Software Eng. Notes*, Oct. 1989, pp. 67-76.
10. J. Noll and W. Scacchi, "Integrating Diverse Information Repositories: A Distributed Hypertext Approach," *Computer*, Dec. 1991, pp. 38-45.



Peiwei Mi is a PhD candidate in the computer science department at the University of Southern California. His research interests include knowledge-based systems to support the software process, process-driven software-engineering environments, organizational analysis of systems-development projects, and distributed problem-solving.

Mi received a BS and an MS in computer science from the University of Science and Technology of China.

Walt Scacchi is an associate research professor in the decision systems department at USC. He created and directs the USC System Factory Project. His research interests include very large scale software production, knowledge-based systems for modeling and simulating organizational processes and operations, CASE technologies for developing large heterogeneous information systems, and organizational analysis of systems-development projects.

Scacchi received a PhD in information and computer science from the University of California at Irvine. He is a member of the ACM, IEEE, the American Association for Artificial Intelligence, Computing Professionals for Social Responsibility, and Society for the History of Technology.

Address questions about this article to Scacchi at Decision Systems Dept., USC, Los Angeles, CA 90089; Internet scacchi@pollux.usc.edu.