Tool integration in lifecycle support environments

by JAYASHREE RAMANATHAN*

Universal Energy Systems
Columbus, Ohio

and

VASUDEVAN VENUGOPAL

Ohio State University
Columbus, Ohio

ABSTRACT

Two pragmatic requirements are placed on lifecycle support environments: (1) one must be able to integrate existing tools into the environment, and (2) the environment must possess an open-ended architecture. The approach must therefore consider the large number of tools that support various phases of a lifecycle. The diversity of such tools makes them hard to integrate into an environment such that they can operate in a coordinated manner and can communicate with each other.

This paper provides a possible approach to the tool integration problem in which the environment architecture and user interface issues are also taken into account. It is shown herein that such an approach leads to a very general and powerful technique of integrating tools. Apart from being able to handle evolution both in the environment and in tools, the approach allows the enforcement of policies on tool invocation and on tool operation.

*The author currently is on leave of absence from Ohio State University.
INTRODUCTION

This paper addresses the issue of integrating diverse tools into lifecycle support environments. The integration problem is examined in the context of the following architectural characteristics:

- A logical database serves as the repository for all project information.1-3
- The database is object-oriented and its conceptual model allows the representation of all objects involved in the lifecycle (for example, objects representing people such as programmers, generated objects such as source code, and derived objects such as object code) and their interrelationships (like the "owns" relationship between a programmer and a code object or a "part_of" relationship between an object describing the project and another describing a programmer involved in the project).2
- Tools that are integrated into this environment access existing objects (by means of views) and tool products are distributed into the database in a manner consistent with the conceptual model.
- The environment allows diverse types of existing tools to be integrated regardless of the idiosyncrasies of their operation. It also allows new tools to be added without any major modifications to the tools.
- The environment has at least a rudimentary notion of an activity and such relationships between activities as sequencing and concurrency.3-5

The first two characteristics are widely accepted among software environment builders. The next two are unique to the environment described herein; however, they are essential pragmatic requirements for a real-world environment. Given the breadth of activities that such an environment supports and the diverse tools that are involved, it would be impossible to come up with a single representation for all tools or, for that matter, to predict all the tools that would be integrated into the environment. The emphasis on existing tools does not preclude the possibility of building tools for such an environment. As for the last point, any software environment attempting to enforce policies on users must have a conception of software development as a set of related activities. Much of the policy enforcement at a macro level is done by permitting, denying, or constraining specific activities at specified times.

"Loose integration" is proposed here as an appropriate, widely applicable paradigm for tool integration in lifecycle support environments. Broadly speaking, loose integration is a view-oriented mechanism for bidirectional communication between the tool and the database through input-extraction and output-distribution views.

The distinctness of this approach stems from the fact that tool integration is treated as an issue to be examined in the larger context of modeling the software process. It does not treat tool integration as an issue of tool communication and therefore independent of the process model.6,7 Further, it does not treat the process model as being, in some sense, a "merging" of tool views.8 Such an approach would subsume the model aspect as one necessary for tool communication rather than as a separate issue.

Listed below are some advantages of the loose integration scheme proposed here. Many of these advantages accrue because of the unified treatment of tool integration with other environmental issues.

- **Encapsulation:** When integrating a tool, you need not be aware of the other tools in the environment.
- **Tool communication:** Communication between tools need not be set up. It happens automatically and indirectly if their input view specifications intersect (i.e., the two view specifications refer to the same object(s)).
- **Tool coordination:** Since tool invocation is an activity, the activity model can coordinate this tool's invocation with others in much the same way as it coordinates any other activities. Thus, tool coordination is merely a special case of activity coordination.
- **Enforcement of policies on tool invocation:** Policies such as automatic invocation of tools, constraints on their invocation, and so on can be implemented using the same mechanism that is used to monitor activities (the activity model) and enforce system policies (using daemons described in the next section).
- **Increased tool versatility:** Since the same tool can be associated with many views, it can be used in many ways. For example, one view specification may extract the view for compiling an entire project while another might be meant to let an individual programmer compile his or her module. Both views would be associated with the same compiler tool.

Some other advantages are better illustrated by specific examples presented in a later section.

The next section briefly digresses to a description of the object model that is used for the software process. This description is brief and is added only to facilitate an understanding of the examples that follow. Later sections describe and illustrate loose integration of tools with examples and discuss the scope and limits of its applicability. Lastly, the status of our implementation is discussed.
THE OBJECT MODEL

Information Frames (see Figure 1) are used to model all the data objects involved in the lifecycle. Information frames correspond to physical and logical entities in the software process. Slots of a frame describe properties of an object. Some slots link the parent frame to other frames or slots. The link type represents a particular relationship between participating frames. Slots and frames may possess attributes that serve as variables for recording local state and history information as well as protection and display information.

The real difference between information frames and structured objects of other object-oriented models lies in the attachment of procedures to slots or frames. These procedures (called daemons) are triggered (as against being explicitly invoked) under a variety of environmental conditions. These conditions could be user and tool operations on slots and frames (such as Visit, Modify, Exit and so on), the occurrence of certain environment states (such as design in progress), or change in environment states (such as design completed), to name a few. The variety of triggering conditions allows daemons to help in a variety of activities such as assisting a user, maintaining data consistency in an object base and enforcing policies on users and tools. Most of the policy enforcement by daemons is of an "all or none" nature. An ongoing activity or subactivity could trigger a daemon. Successful execution of the daemon validates the activity. If the activity or subactivity violates some environment policy, an exception is raised during the execution of the daemon leading to a rollback of the errant activity. The environment is then reverted to the most recent consistent state.

A frequently used display view for frames is called a form. Forms consist of panels which correspond to the underlying slots of a frame. A user navigates through the object base by moving between text panels and down links to other frames. Any daemons related to user focus are fired as the user navigates through the database. The commands that a user can issue when in a panel are controlled by a menu. Forms are, however, only one of many possible unpars schemes for a frame.

CONCEPTUAL ISSUES IN TOOL INTEGRATION

The decision to integrate a tool is not a mere data conversion issue. Integrating a tool involves resolving such issues as the nature of the user-tool interface, the tool's input and output views, and the policies one wants to enforce on tool usage. Provided here is a classification of the kinds of knowledge involved in integrating a tool.

1. Interface-oriented knowledge:
   a. Commands involved in tool invocation. (That is, should the tool be invoked by users using menu selection, control keys, or some other scheme? Should the user's focus be inferred from the cursor position, mouse position, or by some other mechanism?)
   b.Granularity of data. (That is, on what chunk of information should the tool be applied, and how does this relate to the user focus when invoking the tool?)
   c. Granularity of control. (That is, should the command provided to users constitute one tool command or should one design user commands that are implemented as combinations of underlying tool commands? Also, how and when should users get back control and how should users be notified of and made to look at tool outputs and diagnostics?)
   d. Interface-oriented daemons. (That is, is there a need to include consistency-checking daemons beyond whatever was implemented in the object-model? For example, completeness and validity of user-supplied data can be ensured by associating daemons with data panels that monitor the data even as it is being keyed in.)

2. Tool-oriented knowledge:
   This comprises of knowledge needed to reconcile the tool and information-base disparity. To do this, it is necessary to specify the input-extraction and output-distribution views as well as parse-up and parse-down procedures between the tool and the information base. The view definition language must have primitives not only for specifying network traversal but also for specifying view modifications, view dependencies, and exception handling primitives to handle exceptional situations that might occur during traversal.

3. World-oriented knowledge:
   Concurrently with the integration of a tool, the system may need to enforce tool-related policies. For example, along with the integration of a compiler to allow compilation of modules, it may be desirable to enforce the protocol to automatically inform the project manager of a successful compilation. To do so requires knowledge about other objects in the "world-model" of the enterprise such as the manager's frame. Implementing the protocol may also require knowledge about what the manager is to be informed and how. Much of this would depend on personnel hierarchy and system policy about protection and distribution of information. Generally some world-oriented knowledge is required about the tool's role in the enterprise as well as policies and protocols relating to tool application.
EXAMPLES

1. **Tool**: Editor

   **Assumed tool characteristics:**
   - Interactive nature of the tool.
   - Works on buffers.

   **Desired interface behaviour:**
   The user should be able to use the editor in a panel in much the same way as he edits a buffer.

   **Steps in integrating such a tool:**
   - Associate a buffer with each panel.
   - Intercept user commands and pass them on to the editor with the buffer context.
   - Redisplay the updated buffer on the affected panel.

   **Policies that can be implemented:**
   - Sense the language in which the user is coding (on any given coding panel) and put the editor in the appropriate model (e.g., C-mode in EMACS \[^{10}\] if the user is coding in C). This assumes that the editor provides language sensitive assistance.
   - Automatically save changes after every 10 commands.

2. **Tool**: RCS (Revision Control System)\[^{11}\]

   **Salient tool characteristics:**
   - Batch tool
   - Not a data transformer but a repository for textual information.

   **Some policies that can be implemented:**
   a. Maintain "revisions" of any slot that contains more than 500 lines of text.

   **Implementation:**
   - Associate a linecount attribute with any slot containing text that stores the number of lines in the slot (this could be computed every so often).
   - Have a daemon associated with each such slot that fires whenever the user exits the associated panel, and if the panel is larger than 500 lines, have checks in the slot text to RCS (the slot can be uniquely identified by the frame-id/slot-heading pair).

   RCS handles the versioning of slots, allowing us to implement such environment policies as:
   b. Allow the owners of the participating frame to decide when they want to store a modified text as a new revision and to retrieve any particular version.

   **Implementation:**
   - Tie the "check in a new version," "check out version," and "create a new version" commands to keystrokes and invoke in the context of the panel that is the user-focus.
   c. In a project, only the chief programmer has rights to create new revisions. Subprogrammers are not to be aware of the underlying versioning.

   **Implementation:**
   - Have a daemon that is triggered when the chief programmer visits his frame.
   - This daemon should augment the displayed menu with "create a new revision" as an invocable command.
   - If the chief programmer issued this command, traverse each of the "subprogrammer" links out of this frame and check in the "source code" slots of these frames for a new revision.
   - The name of the RCS file associated with the "source code" slots of the subprogrammers is stored as an attribute of the subprogrammer frame.
   - Whenever any subprogrammer visits his frame, display it in the usual manner except to check the text associated with the "source code" slot and display it in the corresponding panel.

3. **Tool**: Compiler

   **Salient tool characteristics:**
   - Batch tool
   - Data transformer

   **Mode of integration:**
   Given that the coding frame has the form shown in Figure 2 and that "uses" and "subprogrammer" links exist between such frames, it is desired that the chief programmer be provided with a single command to compile the entire project's code. Any error messages are to be distributed to the appropriate places.

   **Some comments:**
   Shown below are the input and output views for integrating a compiler in the manner mentioned above. The following aspects of the problem and the working of the view-interpreter are worth noting:
   - There is only an implicit ordering among the subprogrammer modules. The view specification includes exception handlers that cause backtracking until the modules are incorporated into the view in the right order.
   - The contribution of any frame instance depends not only on the type of the frame but also on the contents of some panels in the frame instance. Each unique selection criterion forms and "instance view." Many such instance views combine to form the input view for the tool (com-

---

Figure 2—The view for "compile_project"

---

From the collection of the Computer History Museum (www.computerhistory.org)
Compiler view:

Chief programmer instance view
begin view
frame instance selection:
(heading(slot) = "title") and
(contents(slot) = "chief programmer");
slot selection:
(type(slot) = "LINK") and
(heading(slot) = "subprogrammer") and
(untraversed_link(slot)) → traverse_link(slot);
heading(slot) = "declaration"
or (heading(slot) = "code")
→ add_to_view(slot);

exceptions:
forall(((heading(slot) = "declaration") or
(heading(slot) = "code")) and
(contents(slot) = NIL) → handled by foo;
forall((type(slot) = "LINK") and
(contents(slot) = NIL) → handled by foo;

terminate condition:
forall((type(slot) = "LINK" → traversed(slot))
end view

Subprogrammer instance view
begin view
frame instance selection:
(heading(slot) = "Title") and
(contents(slot) = "subprogrammer");
slot selection:
(type(slot) = "LINK") and
(heading(slot) = "subprogrammer") and
(untraversed_link(slot)) → traverse_link(slot);
heading(slot) = "declaration" or
(heading(slot) = "code")
→ add_to_view(slot);

exceptions:
forall(((heading(slot) = "declaration") or
(heading(slot) = "code")) and
(contents(slot) = NIL) → handled by foo;
forall((type(slot) = "LINK") and
(contents(slot) = NIL) → handled by foo;
forall((type(slot) = "LINK") and (heading(slot) = "uses")
→ is_visited(object_pointed_by(contents(slot)))
handled by untraverse;

end view

end combine

output_distribution_view for compiler
begin dist
associate(error_message, line_text)
using assoc_procedure;
insert_into_slot(find_slot_in_view(line_text, slot_num, location),
error_message, location + 1);
end dist

4. Combinations of tools:

Policies and activities requiring combinations of tools to be
invoked in a coordinated manner is handled in one of two
ways. In the case that the coordination is complex, it is han­
dled in the activity model which treats each tool invocation as
an activity and models the entire coordinated transaction as a
petri net. Simple and hardwired interactions between tools
can be implemented using daemons.

SCOPE AND LIMITATIONS

The ease of loose integration of any tool depends on the ease
with which the environment can mediate and arbitrate be­
tween the tool and the user. In almost all batch tools, the
semantics of tool commands and the nature of input required
by the tool are quite well defined. Normally, user commands
correspond closely to single tool operations. It is therefore
straightforward to integrate batch tools into the environment.
The integration is more graceful if the underlying oper­
ating system provides "virtual screen" or "pseudoterminal" (UNIX)
facilities so that tool diagnostics and messages that are
directed to the screen can be intercepted and the environ­
ment can decide what it wants to do with them. In an oper­
ating system devoid of these facilities, the environment cannot
filter out these diagnostics but it can still restore the earlier
screen status once the tool has finished executing.

Interactive tools that interact through textual data and com­
mands are integrated by using a pseudo-terminal interface to
the tool. The environment arbitrates by isolating the inter­
active tool from the physical screen. Each command during an
interaction is treated like one invocation of a batch tool with
that command. The results of the interactive command are
captured by the pseudo-terminal interface and relevant parts
presented to the user in the right context.

For interactive tools in which the interaction involves both
textual and graphical information as well as the use of pointing
devices (e.g., mouse or light pen), the very advantage of such
tools causes problems in integration. The fact that a user can
point at any object on the screen makes it harder for the
environment to control user actions than it was in the case
where the system could track or control cursor motion. The
fact that simple graphical figures can represent textual infor­
mation in a greatly condensed form leads to the problem that
for any graphical figure, the underlying frame representation
(which is essentially textual) can be very complex. Moreover,
incremental changes to the graphical representation may lead
to major changes in the underlying frame representation due to the difficulty in translating the graphical context of the change into an underlying frame context and because we cannot guarantee that there exists a mapping between an arbitrary set of graphical operations and underlying frame operations that ensures consistency under composition. For example, the effect of adding a link between an SADT \textsuperscript{12} box A and another box B can be understood only in the context of other boxes in the diagram.

IMPLEMENTATION

The paradigm of loose integration has been examined and tested as part of the TRIAD project in progress at The Ohio State University and the KI shell being developed at Universal Energy Systems (see \textsuperscript{32} for a detailed description of the environment architecture). Some examples include the integration of a C compiler and the dbx debugger as well as the integration of synthesized tools such as a tool for providing graphical views of project information. The TRIAD shell has also been used to bring up process support environments for non-software lifecycles allowing the opportunity of integrating a variety of design and simulation tools used in manufacturing an expert system as well as numerous other domain-specific tools. The language framework for describing views is under development.

CONCLUSIONS

Given the trend in software environments towards having a common database for storing project information, we examine a view-oriented approach to tool integration for permitting bidirectional communication of information between a tool and the database. We also believe that a carefully designed object-oriented architecture suitable for lifecycle support goes a long way in solving the problems that are typically encountered in integration. A common interface for database access and tool invocation as well as an active database in which user actions and changes to data are monitored as they go on eases policy enforcement and leads to significant enhancements in the ways in which an already existing tool can be used on incorporation into the environment.

REFERENCES
