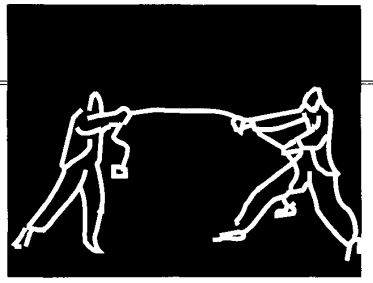


Definitions of Tool Integration for Environments

IAN THOMAS, *Hewlett-Packard*
BRIAN A. NEJMEH
Innovative Software Engineering Practices

◆ *What does "integration" mean? Integration is a property of tool interrelationships. Understanding it will help us design better tools and integration mechanisms.*

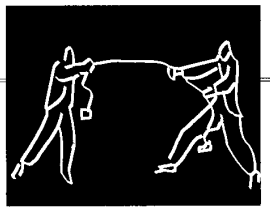
There has been considerable discussion in recent years about the integration of software-engineering environments, perhaps beginning with the use of "integrated" in the term IPSE (integrated project-support environment) and continuing with the coinage of the terms ICASE (integrated CASE) and ISEE (integrated software-engineering environment).

Although some have tried to define precisely what "integration" means in these terms, we believe these definitions are not as precise as they should be. We believe integration is not a property of a single tool, but of its relationships with other elements in the environment, chiefly other tools, a platform, and a process. Of these, we believe the key notion is the *relationships* between tools and the properties of these relationships.

Tool integration is about the extent to which tools agree. The subject of these agreements may include data format, user-interface conventions, use of common functions, or other aspects of tool construction. To determine how well tools agree — and how well they are integrated into an environment — we propose a framework that focuses on *defining* integration, independently of the mechanisms and approaches used to support integration.

Our purpose is to identify the goals of integration and propose some questions that establish what information is needed to know that these goals have been reached. In this respect, we are following Victor Basili and David Weiss's approach for metrics development, in which they advocate identifying goals, questions that refine the goals, and quantifiable metrics

0740-7459/92/0300/0029/$03.00 © IEEE



that provide the information to answer the questions.[1] However, we do not propose quantifiable integration metrics here.

Anthony Wasserman identified five kinds of integration: *platform*, which is concerned with framework services; *presentation*, concerned with user interaction; *data*, concerned with the use of data by tools; *control*, concerned with tool communication and interoperation; and *process*, concerned with the role of tools in the software process.[2]

We extend Wasserman's analysis by building on his definitions of presentation, data, control, and process integration. Our elaborations are based on experience with framework services and integrated environments and an analysis of the issues. Because our focus is on the relationship among tools, we do not consider platform integration, which we regard as providing the basic elements on which the agreement policies and usage conventions for tools are built.

There are two viewpoints in the discussion of integration: the environment user's and the environment builder's.

TWO POINTS OF VIEW

There are two points of view in the discussion of integration: the environment user's and the environment builder's. The environment user is concerned with perceived integration at the environment's interface. The user desires a seamless tool collection that facilitates the construction of systems on time and within budget. The environment builder, who assembles and integrates tools, is concerned with the feasibility and effort needed to achieve this perceived integration.

The user would like to see well-integrated tools; the builder would like to see easily integrable tools. Both perspectives are important, and many of the integration properties we describe are meaningful from both points of view.

Our more precise way of looking at integration has proved useful to four groups: users, tool evaluators, tool writers, and framework-technology builders. It provides them with a definitional framework in which

- users can characterize areas in their environments in which tools should be better integrated;
- tool evaluators can identify criteria to evaluate tool sets they want to include in an integrated environment;
- tool writers can examine design and architectural issues as they develop the next generation of integrated environments and identify good practice for the use of emerging integration-support mechanisms; and
- framework-technology builders can explain how proposed and existing integration mechanisms contribute to improvements in integration in terms of the properties described here.

We have tried to separate integration properties so as to identify them as clearly and independently as possible. In practice, we know that tool writers can use a single integration-support mechanism to improve several integration properties.

TOOL INTEGRATION

The goal of a software-engineering environment is "to provide effective support for an effective software process."[3] We believe support is more effective if the environment is integrated — if all its components function as part of a single, consistent, coherent whole.

Integration means that things function as members of a coherent whole. When we say, "*A* is well integrated with *B*," we are really making many statements because *A* and *B* are composites with many characteristics. To understand how well *A* is integrated with *B* requires a careful examination and comparison of each characteristic.

We extend Wasserman's four kinds of integration by identifying several well-defined properties that characterize the various integration relationships between tools. The software-engineering community generally agrees on the importance of these four tool relationships, but the definitions of integration properties are not as precise as they should be.

Figure 1 shows an entity-relationship diagram depicting a single tool, four relationships, and our elaborated properties for each relationship. The four well-known relationships are

- *Presentation*: The goal of presentation integration is to improve the efficiency and effectiveness of the user's interaction with the environment by reducing his cognitive load.
- *Data*: The goal of data integration is to ensure that all the information in the environment is managed as a consistent whole, regardless of how parts of it are operated on and transformed.
- *Control*: The goal of control integration is to allow the flexible combination of an environment's functions, according to project preferences and driven by the underlying processes the environment supports.
- *Process*: The goal of process integration is to ensure that tools interact effectively in support of a defined process.

Our approach uses binary relationships, which raises the issue of whether integration should be defined as how well *two* tools are integrated or if it should be defined as how well *many* tools are integrated. We believe that the second definition can be adequately captured as an aggregate property, derived from how well individual tool pairs are integrated. However, our focus on how well two tools are integrated does *not* mean that we support integration mechanisms that allow "private" tool agreements. It is important to distinguish a *definition* of integration from a good *mechanism* for integration support.

Presentation integration. The goal of reducing a user's cognitive load should apply to individual tools, tool sets, and the environment as a whole. It can be achieved by letting users reuse their experience in interacting with other tools by

- reducing the number of interaction and presentation paradigms in the environment,

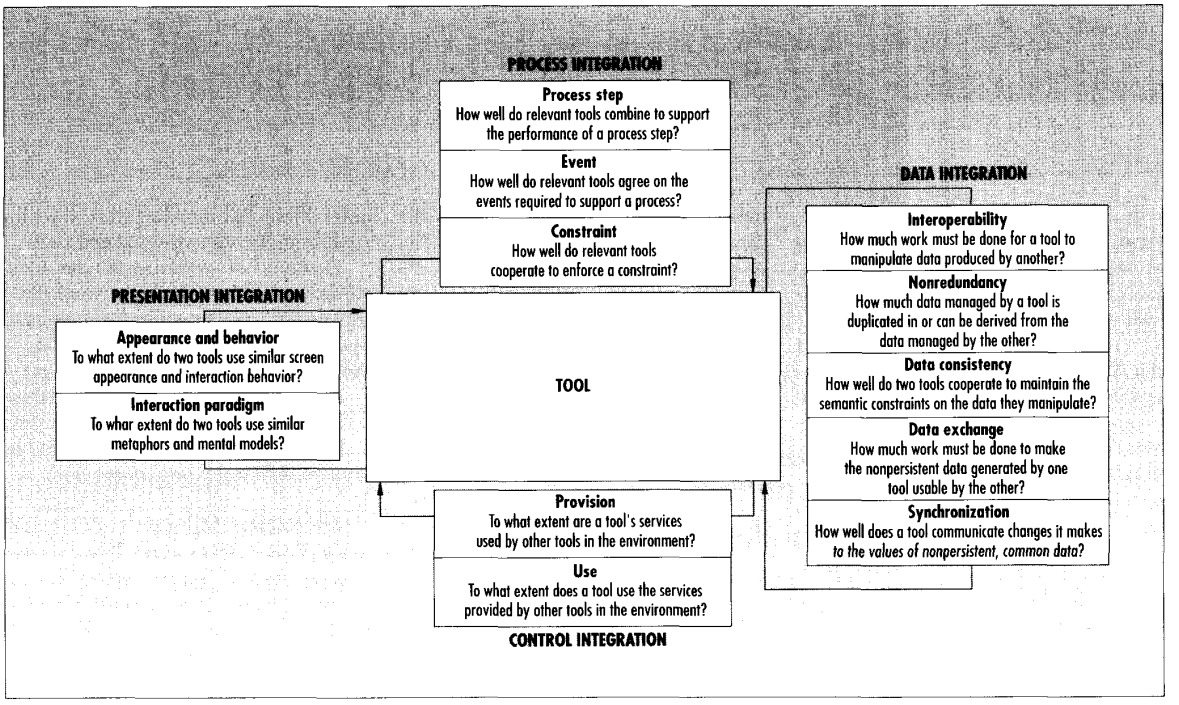


Figure 1. Entity-relationship diagram depicting a single tool, four relationships, and elaborated properties for each relationship.

- providing interaction and presentation paradigms that match the user's mental models,
- meeting the user's response-time expectations, and
- ensuring that correct, useful information is maintained at the disposition of the user.

We identify two properties in this class, which we base on the relationships between the user interfaces of two tools: appearance and behavior integration and interaction-paradigm integration.

Appearance and behavior. This property answers the question, How easy is it to interact with one tool, having already learned to interact with the other? In other words, how similar are the tools' screen appearance and interaction behavior?

Two tools are said to be well integrated with respect to appearance and behavior integration if a user's experience with and expectations of one can be applied to the other.

Appearance and behavior integration captures the similarities and differences between the *lexical* level of the two tools' user interfaces — how the mouse clicks, the format of the menu bars, and so on. It also covers some aspects of their *syntactic* level differences and similarities — the order of commands and parameters, uniformity of the presentation of choices in dialogue boxes, and so on.

Both Motif and OpenLook specify compliance levels that are relevant here. However, both leave aspects of appearance and behavior undefined, which may lead to unnecessary, confusing differences between the appearance and behavior of two tools.

A broader definition of appearance and behavior should also cover response-time aspects. Similar interactions with two tools should have similar response times for them to be well integrated with respect to this property. Appearance and behavior might also include using a common meaning for verbs and commands.

Interaction paradigm. This property answers the question, "How easy is it to interact with one tool having already learned the interaction paradigm of the other?" In other words, to what extent do two tools use similar metaphors and mental models to minimize learning and usage interference?

Two tools are said to be well integrated with respect to interaction-paradigm integration if they use the same metaphors and mental models.

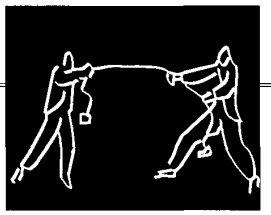
Clearly, it is important to balance the use of one versus many metaphors. A single metaphor may be awkward or ill-adapted for some cases, while using many metaphors may provide one that is well-suited for each case but makes it difficult to transfer experience between tools.

For example, a tool to access and browse a database may use a filing-system metaphor, with filing cabinets, drawers, dossiers, and so on. This metaphor might impose a strict containment relationship between cabinets and drawers and between drawers and dossiers. Another tool for accessing the same information may present a different metaphor that involves navigating around a hypertext structure, with no emphasis on containment relationships.

A user who must use both tools may be confused by the different navigation metaphors, so we would say that these two tools are not well-integrated with respect to their interaction paradigms.

Data integration. The information manipulated by tools includes persistent and nonpersistent data (which does not survive the execution of the tools that are sharing or exchanging it). The goal is to maintain consistent information, regardless of how parts of it are operated on and transformed by tools.

Data integration between two tools is not



relevant when they deal with disjoint data. Also the data-integration properties should reflect the use of the "same" data by tools, even if that data is represented differently or can be deduced from other data.

We have identified five properties in this class, defined between the data management/representation aspects of two tools: interoperability, nonredundancy, data consistency, data exchange, and synchronization.

Interoperability. Suppose a tool using some data in the environment has a certain view of that data. Another tool may need to use some or all of the data used by the first tool but may have a different view of that data. How views differ can range from the peculiarities of character representations to differences in the information model each tool assumes and captures in schemas. Two tools may even assume different semantics for data stored using the same schema.

The interoperability-integration property answers the question, "How much work must be done to make the data used by one tool manipulable by the other?" In other words, what must be done for the two tools to see the data as a consistent whole?

This property illustrates the different viewpoints of the user and builder. The environment user may perceive that two tools are well integrated because they present a common schema at the user interface, even if each tool actually has a different data model. The environment builder will see that the tools are not well integrated, because they use different data models and require a lot of work to make the data of one tool manipulable by the other.

Two tools are said to be well integrated with respect to interoperability integration if they require little work for them to be able to use each other's data.

For example, suppose a design tool produces data in a certain format and that data must be manipulated by another tool that expects data in a different format. To make it available to the second tool, the data must be run through a conversion program. From the environment builder's point of view, these two design tools are not as well integrated with respect to interoperability integration as two tools that use the same format and model. If the user must initiate the conversion, the tools are also not well integrated from the user's point of view.

Tools are well integrated when they have a common view of data.

Tools are well integrated when they have a common view of data. Some environments achieve this common view by using common internal structures for the information they manipulate (such as programs and design artifacts). Examples include the Interlisp environment and Rational's Ada environment. Other environments use common schemas. Examples include Pact[4] and IBM's AD/Cycle Information Model.[5]

Nonredundancy. This property answers the question, "How much data managed by a tool is duplicated in or can be derived from the data managed by the other?" In other words, it identifies the redundancy in the data that the two tools independently store and manipulate.

Two tools are said to be well integrated with respect to nonredundancy integration if they have little duplicate data or data that can be automatically derived from other data.

Redundant information in a database, whether duplicate or derived, is undesirable because it is difficult to maintain consistency. Nonredundancy is relevant even if the tools store their data in the same database.

Well-integrated tools should minimize redundant data. For example, suppose two project-management tools, which operate on the same data, use the concept of a project task. One assumes that tasks have two attributes, start_date and end_date; the other assumes that tasks have two attributes, start_date and duration. Although both tools use the same start_date, they are not well integrated from the builder's point of view because duration can be derived from the start_date and end_date.

Designers of environments that must maintain consistent duplicate and derived data must choose strategies for timing the updates to the duplicate and derived data — the well-known "refresh-time" problem.

Avoiding duplicated data in a database still allows the use of *replicated* data, which a distributed database might provide to improve robustness or performance.

Data consistency. Maintaining the consistency of duplicate or derived data is a frequently observed special case of how to maintain some general semantic constraint on the data. For example, a designer may want to ensure that the sum of certain attributes in the database is less than some value: Tool 1 manipulates some data *A*, Tool 2 manipulates some data *B*, and there is a semantic constraint relating the permissible values of *A* and *B*.

The data-consistency property answers the question, "How well do the tools indicate the actions they perform on data that is subject to some semantic constraint so that other parts of the environment can act appropriately?" In other words, how well do two tools cooperate to maintain the semantic constraints on the data they manipulate?

Two tools are said to be well integrated with respect to data-consistency integration if each tool indicates its actions and the effects on its data that are the subject of semantic constraints that also refer to data managed by the other tool.

Data exchange. Two tools may want to exchange data. The data may be in the form of initial values communicated from the first tool to the second when it starts execution or it may be in the form of values passed to a tool while it is executing. To exchange data effectively, the tools must agree on data format and semantics.

This property answers the question, "How much work must be done to make the data generated by one tool usable by the other?" In other words, what must be done to make the data generated by one executing tool manipulable by the other?

Two tools are said to be well integrated

with respect to data-exchange integration if little work on format and semantics is required for them to be able to exchange data.

This definition is very similar to the interoperability definition, but data-exchange integration also applies to nonpersistent data.

Some environments may allow the environment builder, rather than the tool designer, to decide which data is persistent. We make the distinction between data-exchange and interoperability because the mechanisms that support data-exchange integration may be different from the mechanisms that support interoperability integration, although some environments may use the data-management system to implement interoperability integration. The definition of persistent data is independent of whether or not the data is stored in a database. Data-exchange integration is also relevant whether or not the two tools use the same database.

Suppose you have two project-management tools. The first is a scheduling tool that displays a network of project tasks, each represented as an icon. Each task icon has a user-defined name represented as a task-type attribute in the environment's database. The second tool is a time sheet that lets the user record time spent on project tasks. The user interface to the second tool is a spreadsheet, with each column headed by a task identifier that is used as an account number for charging. This task identifier is also stored as a task-type attribute in the environment's database.

Now suppose the user wants to copy a task icon from the scheduling tool and paste it into the task-identifier cell of the time-sheet tool, which would then display the task identifier of the task icon. Clearly, both tools must agree on the format and semantics of the data exchanged via the copy-and-paste mechanism.

Synchronization. In general, a single tool will manipulate some persistent data and will also need nonpersistent data for execution. This is also true of a set of cooperating tools, which typically will manipulate persistent data (and interoperability, nonredundancy, and data-consistency properties determine how well integrated they are to do this). Cooperating tools may also need to maintain the consistency of the nonpersistent data that may be replicated in several tools in a set.

The synchronization property answers the question, "How well does a tool communicate changes it makes to the values of nonpersistent, common data so that other tools it is cooperating with may synchronize their values for the data?" Two tools are well integrated with respect to synchronization integration if all the changes to all shared nonpersistent data made by one tool are communicated to the other.

Suppose a set of development tools includes a debugger and a browser. The set might have been constructed with the idea that the debugger and browser would share a single current_source_code_line position. When the debugger stops at a breakpoint, it changes the current_source_code_line position, which the browser uses to display the lines above and below current_source_code_line. Both tools need a consistent view of this datum and must inform each other of changes they make to it.

This definition is very similar to data-consistency integration, except it applies to nonpersistent data. We make the distinction because the mechanisms used to support synchronization may be different from the mechanisms used to support data consistency. Both mechanisms must define a refresh time to communicate changes between tools.

Control integration. To support flexible function combinations, tools must share functionality. Ideally, all the functions offered by all the tools in an environment should be accessible (as appropriate) to all other tools, and the tools that provide functions need not know what tools will be constructed to use their functions.

For tools to share functionality, they must be able to communicate the operations to be performed. Because operations require data, the tools must also communicate data or data references. In this regard, control integration complements data integration.

Data integration addresses data representation, conversion, and storage issues; control integration addresses control-transfer and service-sharing issues. We have identified two properties, defined on the control relationship between two tools: provision and use.

Provision. This property answers the question, "To what extent are a tool's services used by other tools in the environment?"

A tool is said to be well integrated with respect to provision integration if it offers services other tools in the environment require and use.

Suppose you are building a project-management tool that requires the user to enter a brief description of project tasks. Such a tool requires an editing service to enter textual task descriptions. In this case, provision integration refers to the extent to which the editing tool provides the services required by the project-management tool.

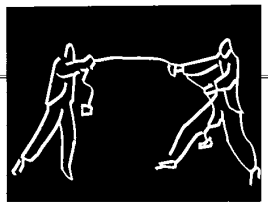
Achieving high use integration requires modular tools.

Use. This property answers the question, "To what extent does a tool use the services provided by other tools in the environment?"

A tool is well integrated with respect to use integration if it appropriately uses the services offered by other tools in the environment.

Achieving high use integration requires modular tools. A tool written without regard for replacing services that it provides with similar services will not achieve good use integration. The same tool could be written so that it either expects certain services in its execution environment and provides a means of incorporating such services into the tool or provides a convenient way of replacing a tool-provided service with a comparable service offered by another tool in the environment.

The project-management tool in the earlier example would be highly inte-



grated with the editing tool with respect to use integration if it used the editing tool's services. The project-management tool would then have to provide a convenient way to replace its text-editing services with those offered by the editing tool.

Process integration. There are three dimensions to ensuring that tools interact well to support a defined process:[6]

- A *process step* is a unit of work that yields a result. Assessing design performance is a process step.
- A *process event* is a condition that arises during a process step that may result in the execution of an associated action. Conducting a successful compile may result in the scheduling of a unit test.
- A *process constraint* restricts some aspect of the process. A constraint might be that no person can have more than 10 process steps assigned to them concurrently.

A tool embodies a set of assumptions about the processes in which it may be used; two tools are well integrated with respect to process if their assumptions about the process are consistent. The degree of consistency between the process assumptions of tools strongly influences the degree of potential process integration for all the process-integration properties we identified.

Obviously, tools that make few process assumptions (like most text editors and compilers) are easier to integrate than tools that make many process assumptions (like those that support a specific design method prescriptively or generate test skeletons from a design notation).

Whether we need to consider how well two tools are integrated with respect to process integration depends on whether they are both relevant to the same process step. For example, the relevant tools in asessing design performance might be a graphical design tool and a performance-analysis tool. Other tools that may be irrelevant to this process step are the assembler and debugger. Whether a tool is relevant or irrelevant depends on the process property.

We identified three process-integration properties, defined on the process relationship between two tools: process step, event, and constraint.

Processs step. This property answers the question, "How well do relevant tools in the environment combine to support the performance of a process step?"

The performance of a process step will often be decomposed into executions of various tools. Tools often have preconditions that must be true before they can perform work to achieve their goals. A tool's preconditions are satisfied when other tools achieve their goals.

Tools are said to be well integrated with respect to process-step integration if the goals they achieve are part of a coherent decomposition of the process step and if accomplishing these goals lets other tools achieve their own goals.

We should be designing and building tools to take advantage of the integration support mechanisms available.

Tools can be poorly integrated for several reasons. The goals achieved by two relevant tools can be incompatible in that one tool makes it harder for the other tool to achieve its goals. Similarly, the goals achieved by one relevant tool can be incompatible with the preconditions necessary for the other relevant tool to execute.

Suppose a compile-and-debug process step uses a C++ preprocessor, a C compiler, and a debugger. The C++ preprocessor accepts C++ and generates C, which is compiled. The debugger, which is intended to support source-level debugging, uses information generated by the C compiler. It does not know that the C program was generated by a C++ preprocessor.

In this case, the C++ compilation chain and the debugger are not well integrated with respect to process-step integration because the functions of the C++ compilation chain and the debugger do not form a coherent decomposition of the compile-and-debug step that would permit source-level debugging of the C++ program.

Event. This property answers the question, "How well do relevant tools in the environment agree on the events they need to support a particular process?"

There are two aspects to event agreement. First, a relevant tool's preconditions should reflect events generated by other relevant tools. Second, a relevant tool should generate events that help satisfy other relevant tools' preconditions.

Tools are said to be well integrated with respect to event integration if they generate and handle event notifications consistently (when one tool indicates an event has occurred, another tool responds to that event).

Suppose you want to plan and schedule a module unit test. To do so, the process requires that

1. the module be completely developed (for example, it has run through the Unix lint tool cleanly),
2. the module be checked into the configuration-management system, and
3. test personnel be available.

In other words, the notification of all three events is a precondition to unit-test scheduling. In this case, the lint tool might generate an event indicating a module has passed through lint cleanly, the configuration-management tool might generate an event indicating a completed module has been checked into the system, and a resource-availability tool might generate an event indicating that test personnel are available. These tools would be well integrated with the unit-test scheduling tool because they generate the events necessary to satisfy its preconditions.

Events have some similarities with the data-trigger mechanisms that are part of some persistent object-management systems. One major difference is that events may be signaled without changes to persistent or nonpersistent common data; they indicate only that something of interest and relevance to the process has occurred.

Constraint. This property answers the question, "How well do relevant tools in the environment cooperate to enforce a constraint?"

There are two aspects to enforcing a constraint. First, one tool's permitted functions may be constrained by another's functions. Second, a tool's functions may constrain an-

other tool's permitted functions.

Tools are said to be well integrated with respect to constraint integration if they make similar assumptions about the range of constraints they recognize and respect.

At first glance, constraint integration may appear to be the same as data-consistency integration. However, data-consistency integration is concerned with constraints on data values; whereas, constraint integration is concerned with constraints on process states and how such process constraints affect tool functioning. If process-state information, such as the status of various process steps, is modeled and managed in the environment's data-management system, then the mechanisms that support data-consistency integration can also support constraint integration.

Suppose a process constraint is that the same person cannot both code and test a module. In this case, if the resource-allocation tool that assigns a coding task to a person then prohibits the configuration-management tool from letting that person check out the same module for testing, the resource-allocation tool is well integrated with respect to constraint integration with the configuration-management tool.

In this article, we have emphasized definitions of integration properties on relationships between tools rather than the specific integration-support mechanisms.

The designs of integration-support mechanisms and tools do not proceed independently. Integration-support mechanisms are not developed assuming a constant model of how tools are written. Similarly, tools and tool architectures must evolve to take advantage of the new generation of integration-support mechanisms.

Instead of asking how the integration-support mechanisms support tools, we should be asking how to design and build tools so they can best take advantage of the mechanisms available. Brian Nejmeh offers advice on how to do this.[7]

This article describes a model of environment frameworks that identifies framework services, tools that offer other services, data-management services, and so on. Some proponents of object-orientation argue that these distinctions are false and unnecessary — that all services can be provided by operations associated with objects. However, it is still true that some object types are defined as important to the system and that these objects have predefined operations. Also, each object requires execution-engine services and persistent data management for its implementation, in addition to interobject communication. The integration properties we have identified are largely independent of an environment's technology base and are relevant to object-oriented environments.

We believe other environment elements — including the characterization of the software process — have relationships that could be analyzed using this technique. We look forward to extending and refining this set of relationships and properties as our understanding of tool integration grows. ♦

ACKNOWLEDGMENTS

A number of the initial ideas for this work evolved in working-group discussions involving Tim Collins, Kevin Ewert, Colin Gerety, and Jon Gustafson. We also received helpful comments from Frank Belz, Mark Dowson, Anthony Earl, John Favaro, Peter Feiler, Read Fleming, Steve Gaede, Mike Monegan, Dave Nettles, Huw Oliver, Lolo Penedo, Biff Riddle, Ev Shafrir, and Lyn Uzzle. The National Institute of Standards and Technology Working Group on ISEEs' subgroup on integration generated many test cases we used to refine these ideas.

REFERENCES

1. V.R. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Trans. Software Eng.*, Nov. 1984, pp. 728-738.
2. A.I. Wasserman, "Tool Integration in Software Engineering Environments," in *Software Engineering Environments: Proc. Int'l Workshop on Environments*, F. Long, ed., Springer-Verlag, Berlin, 1990, pp 137-149.
3. V. Stenning, "On the Role of an Environment," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1987, pp. 30-34.
4. M.I. Thomas, "Tool Integration in the Pact Environment," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 13-22.
5. V.J. Mercurio et al., "AD/Cycle Strategy and Architecture," *IBM Systems J.*, Vol. 29, No. 2, 1990, pp. 170-188.
6. M. Dowson, B. Nejmeh, and W. Riddle, "Fundamental Software Process Concepts," Ref. No. 7-7, Software Design and Analysis, Boulder, Colo., 1990.
7. B. Nejmeh, "Characteristics of Integrable Software Tools," Tech. Report 89036-N, Software Productivity Consortium, Herndon, Va., 1989.

Ian Thomas works on software-engineering environment infrastructure at Software Design and Analysis, a research and consulting company. While at Hewlett-Packard, he worked on next-generation object-oriented broadcast mechanisms to support control integration and the use of the Portable Common Tool Environment's Object-Management System to support data integration. His research interests include software-engineering environment frameworks, data management for software-engineering environments, environment-integration technology, and software process support.

Thomas received a BS in applied mathematics from the University of Wales, Aberystwyth and an MS in computer science from the University of London. He is a member of ACM and the IEEE Computer Society.

Brian Nejmeh is president of Innovative Software Engineering Practices (Instep), a firm specializing in software process improvement. Before founding Instep, Nejmeh led several groups at the Software Productivity Consortium. He has written widely on software-engineering topics and is the software-engineering editor for *Communications of the ACM*.

Nejmeh received a BS in computer science from Allegheny College and an MS in computer science from Purdue University. He is a member of Phi Beta Kappa, ACM, and the IEEE Computer Society.

Address questions about this article to Thomas at Software Design and Analysis, 444 Castro St., Ste. 400, Mountain View, CA 94041; Internet thomas@sda.com or to Nejmeh at Instep, 13526 Copper Bed Rd., Herndon, VA 22071; Internet nejmeh@instep.com.