Technical Report
CMU/SEI-92-TR-005
ESD-TR-92-5

Issues and Techniques of CASE Integration with Configuration Management

Kurt C. Wallnau

March 1992

Technical Report
CMU/SEI-92-TR-005
ESD-TR-92-005
March 1992

Issues and Techniques of CASE Integration with Configuration Management

Kurt C. Wallnau

Software Development Environments Project

Unlimited distribution subject to the copyright.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1992 by Carnegie Mellon University.

Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

Requests for permission to reproduce this document or to prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTIBILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

List of Figures

List of Tables

Issues and Techniques of CASE Tool Integration with Configuration Management (CM)

Abstract: Commercial computer-aided software engineering (CASE) tool technology has emerged as an important component of practical software development environments. Issues of CASE tool integration have received heightened attention in recent years, with various commercial products and technical approaches promising to make inroads into this difficult problem. One aspect of CASE integration that has not been adequately addressed is the integration of CASE tools with configuration management (CM)—including both CM policies and systems. Organizations need to address how to make CASE tools from different vendors work effectively with an organization's CM policies and tools (in effect, integrate CASE with CM) within the context of the rapidly evolving state of commercial integration technology. This report describes key issues of the integration of CASE with CM from a third-party integrator's perspective, i.e., how to approach the integration of CASE and CM in such a way as to not require fundamental changes to the implementation of the tools or CM systems themselves.

1 Introduction and Background

In recent years there has been a significant penetration of computer-aided software engineering (CASE) technology into organizations of all sizes. The emergence of a robust CASE tool industry has had substantial impact on the nature of software development environments (SDE) and the technology underlying SDEs. Specifically, there is an observable trend away from large, monolithic SDEs centered around a well-defined SDE *framework*, in favor of smaller environments centered around a coalition of CASE tools. While SDE frameworks such as CAIS[1] [14] and PCTE[2] [5] purported to address the tool integration problem, these efforts have not yet found significant commercial acceptance. Some of the reasons for, and consequences of, the resultant dichotomy between framework-based SDEs and CASE coalition-based SDEs are discussed in [10]. For the moment, however, the bottom line is that SDE framework products do not adequately address the problem of integrating commercial off-the-shelf (COTS) CASE tools.

1.1 What is integration?

The question of what integration really means has arisen because of the dichotomy between CASE coalition-based and framework-based SDEs. The classical framework-based SDE view is that integration is achieved through tool use of common, shared services such as data management, configuration management, and process management. This model is inadequate where tools are not constructed *a priori* to a defined framework standard or if the tools can eas-

1. *CAIS*—Common Ada Programming Support Environment (APSE) Interface Set (CAIS).
2. *PCTE*—(A Basis for a) Portable Common Tools Environment.

ily be amended to do so. In the absence of framework standards and alternative models of integration, CASE environments have emerged which are more like an uncoordinated amalgamation of tools than a coherent, integrated system. Wybolt refers to such uncoordinated tools as “islands of automation” [17].

A number of models of integration have been proposed in recent years to help us understand why integration is difficult, and how to achieve integration.The major models are summarized, below. A more detailed discussion of the following themes can be found in [37].

1.1.1 Control, data and presentation integration

The most widely known model of integration is the three-dimensional control, presentation and data integration classification discussed in [38]. This model is significant because it implies that mechanisms can be combined in different ways to achieve different “depth” or “quality” of integration. For example, data integration could be achieved through simple ASCII file transfer, or through more complex object management systems (OMS). The implicit argument is that integration through the use of more sophisticated mechanisms (e.g., OMS and user interface management systems) leads to greater degrees of “goodness.”

One limitation of this model is that it does not provide insight into how various forms of technology can be combined, or, more generally, what interactions occur among control, data and presentation mechanisms. For example, some mechanisms may interfere with each other (e.g., the MIT X Window System and Sun Microsystem SunVIEW is a particularly severe example), while other mechanisms are in fact a composition of more primitive mechanisms (e.g., remote procedure call, a control integration mechanism, is actually implemented with network protocols and data interchange standards, which are data integration mechanisms). Another limitation is that by focusing solely on mechanisms of integration, the model fails to provide tool integrators and tool builders insight into how to integrate tools, or how to design tools that can be integrated into a variety of potential environments.

1.1.2 Adding the process dimension

One aspect of integration that has only recently been addressed is the role that software process plays in determining what tools (or tool functions) should be integrated, and how they should be implemented. While Wassermann alludes to process as an additional dimension in [38], his view of process was quite constrained. Cagan goes farther in [6] by considering several process dimensions: model integration (specifically, CM models of product structure), life cycle integration and task[3] integration.

It may be more natural, however, to consider process (as a whole) to be an orthogonal dimension of integration to that of mechanism. Put another way, we can view software processes as defining a design context, i.e., a set of design requirements, for the design and implementation

---

[3] Cagan refers to task integration as “process integration.” Since his use of the term specifically focuses on task specification and automation, and not on broader software process issues (e.g., life-cycle issues), the narrower term used in this report is justified.

of an integration solution (whose implementation will be expressed in the combination of control, data and presentation mechanisms). For example, it is nearly meaningless to assert that a particular design tool is integrated with a documentation tool. A more meaningful assertion is that the design tool makes use of documentation tool services in order to generate documentation in some standard life-cycle model form (e.g., Mil-STD 2167 document sets). From this example it appears that integration is not just a combination of mechanisms, but their combined use to achieve some (process) objective.

1.1.3 Integration viewed as relationships among integration factors

The observation that integration can be viewed as the properties attached to a set of relationships was first made in [34]. The essential idea is to think of integration in terms of the relationships among the entities being integrated. In the above example of integration of design and documentation tools, the meaning of integration could be described in terms of relationships between software mechanisms and elements of the software process. Thomas and Nejmeh also make a useful distinction between the notions of *well integrated* and *easily integrated*. Well integrated refers to a user's perception of integrated-ness, while easily integrated reflects the level of difficulty (and hence cost) necessary to achieve a specific user perception of integrated-ness. While this view of integration provides a rich descriptive framework, its practical utility awaits the creation of a comprehensive model which can be used by tool builders, environment infrastructure builders and environment integrators.

1.1.4 Characteristics of integrable tools

One attempt to provide pragmatic advice to tool builders and integrators is to examine integration not from a framework perspective, but from a tool perspective. This approach is taken by Nejmeh in [25], and focuses on the characteristics of tools that make tools readily integrated, or difficult to integrate. As will be seen later in this report, tool characteristics play a crucial role in the outcome of tool integration, and an evolving catalogue (or encyclopedia) of tool characteristics may well prove invaluable for tool integrators and tool builders. However, an organizational framework is necessary to avoid a degeneration of tool characteristics into an *ad hoc* collection of proverbs. It may be the case that such an organizational framework needs to be based upon a more concrete architectural model of integration (such as discussed in Section 1.1.5, below).

1.1.5 A three-level model of integration: mechanism, service and process

The models of integration discussed in the preceding sections share common weaknesses. First, while each model has specific descriptive strengths, none of them provides practical insight into the design and implementation of integration solutions, or the comparison and development of integration infrastructure technology. Second, and more seriously, the above models define integration from the perspective of a technology producer (tool and environment builders). It is becoming increasingly clear, however, that technology consumers (i.e., the end-users of CASE tool technology) will need to play a significant role in tool integration.

As a result of our experiments in integrating CASE tools with CM, and our involvement in the Navy Next Generation Computing Resources (NGCR) Project Support Environment Standards Working Group (PSESWG), we have developed a three-level model of integration that draws upon the positive features of the models discussed above, while also providing a basis for a systematic approach to attacking a tool integration problem.

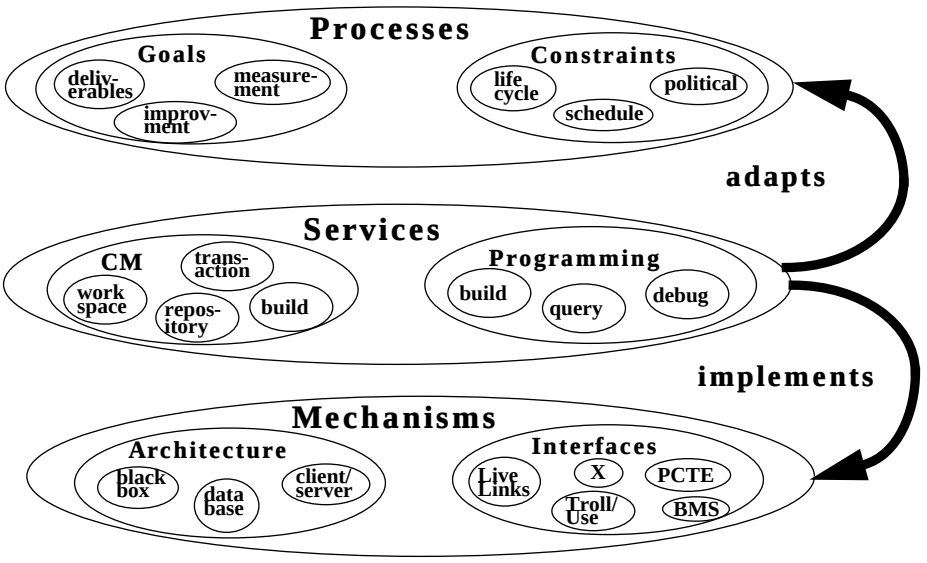

Figure 1-1 Three Levels of Integration

The three-level model illustrated in Figure 1-1 is discussed in more detail in [37]. From the perspective of a tool integrator, the main features of this model are:

- The process level defines the set of requirements which an integration solution needs to support, and hence defines a context in which to carry out an integration design activity.
- The service level defines a layer of abstraction between the requirements and vendor-specific implementation details. This layer of abstraction allows the integration designer to focus on *what* needs to be integrated, rather than on *how* to implement the integration.
- The mechanism level defines the implementation dependencies upon which an integration solution is constructed. By separating the implementation dependencies from the design of an integration solution, it is possible to separate the accidental properties of integration from the essential properties.

Of course, this model is idealized, and in practice an integration design will need to consider mechanisms, services and processes as co-equal contributors to the constraints placed upon

an integration solution. However, it is hoped that this report will illustrate the value of viewing integration from this three-level perspective.

1.2 What is CASE/CM integration?

The integration of CASE with CM is an important topic in its own right, and illustrates well the three-level model of integration just discussed.

First, from a purely pragmatic perspective, the introduction of tools into an environment which provide their own data and configuration management services introduces several sources of CM problems:

- Data stored in separate repositories can introduce data redundancy.
- Different data models may make data sharing impossible.
- Version management (VM) and CM services provided by the tools are frequently linked with private tool data model and data management services.
- Tool VM and CM services are not always delegatable—sometimes these services are an intricate part of tool function (e.g., for multi-user support and build).
- Tool VM and CM services may imply or enforce vendor-specific VM/CM policies.

This short list of potential troublespots by no means exhausts the topic. The difficulties of coordinating and sharing data among egocentric tools, and the introduction of multiple tool repositories makes it difficult to answer such simple CM questions as:

- Who is changing this configuration item?
- How many changes have been made to this configuration item?
- How do I re-establish consistency among these configuration items?
- How do I retrieve and restore version 3.2 of the design, code and test configuration items?

In short, the non-locality and redundancy of data among multiple tool repositories makes it essential that a sound CM strategy be in place to manage the products produced by tools. Part of this strategy must be the integration of the tools with CM services. A CASE/CM integration must address not only the mechanistic aspects of integration (e.g., managing redundant data across heterogeneous tool repositories) but also the process aspects of the integration (e.g., change control policies dictated by a CASE-user's organization).

1.3 Why study "third-party" integration?

While technology producers "own" the mechanism level of integration (in the sense that they have access to source), it is the technology consumers who own the process level. As a result, the technology producers find it difficult to know how to integrate tools since they do not have

sufficiently detailed process requirements.[4] Thus, there is a large gap between the mechanisms of integration and an understanding of the requirements of integration. While it is possible (and desirable) that computer manufacturers and CASE vendors will combine their resources to provide useful and cost-effective CASE integration solutions in the future, no adequate solution has yet reached the marketplace. Thus, in the near term CASE users will need to take responsibility for certain critically important aspects of CASE integration.

In this context CASE users need to be aware of the critical aspects of CASE integration, including those of CASE/CM integration.

1.4 Purpose of this report

The purpose of this report is to identify and discuss the key concepts underlying CASE integration with CM from an end-user's perspective. The underlying hypothesis is that integration is a design activity, and that rather than there being one "right" integration solution, there are many possible solutions. Choosing an integration solution involves understanding process requirements, the CM services available (and their semantics) and implementation constraints, together with making design trade-off decisions related to the integration of these process, service and mechanism concepts. The perspective taken throughout the report is that of a third-party integrator—tool users or systems integration specialists.

1.5 Structure of this report

Section 2 will present the key concepts of CASE/CM integration; the three-level model of integration described in [37] will be used as an organizational framework. Section 3 will illustrate some of these key concepts via a series of abstract integration scenarios. These scenarios illustrate the effect of combining the concepts discussed in Section 2. Section 4 provides a short discussion of some issues likely to arise when considering the integration of CASE with sophisticated CM systems. Section 5 summarizes the key points of the report and indicates future work. Finally, Appendix A provides a detailed description of an integration experiment involving a complex CASE tool and sophisticated CM system.

---

4. This is true even of "CASE Coalitions" formed from strategic CASE vendor alliances. See [10] for an elaboration of the limits of CASE Coalitions.

2 Key Concepts of CASE/CM Integration

The integration of CASE tools involves more than merely identifying (and “integrating”) low-level software mechanisms such as programmatic interfaces and data interchange formats. Other factors, such as software process and tool architectures (among others), also need to be considered. As a result, CASE integration involves a design process whereby solutions emerge as a consequence of design trade-off analysis. That is, there is no “right” integration solution, but rather different solutions which may be applicable under different circumstances. A simple illustration will help make this point.

2.1 Illustration of process, service and mechanism interactions

Consider a hypothetical integration of a CASE tool which provides its own repository and workspace services with a CM system which also provides some support for managing developer workspaces (see [13] and Appendix A of this report for a more detailed description of these services). An obvious issue that must be addressed is when, and under what circumstances, data held locally in the CASE tool’s repository should be “exported” to the CM workspace. A number of possibilities exist. For example, it is possible to export the changes:

- as they are made
- at the end of each user session
- at designated times, e.g., weekly dumps
- at designated events, e.g., major or minor releases
- only at the end of a life-cycle phase, e.g., coding phase

The selection of any one (or more) of these alternatives will be dependent upon factors related to process policies and constraints, the logical services provided by tools and intrinsic implementation characteristics of the tools and CM systems themselves. For example, if the developer workspace supported by the CM system provides access to the workspace by only a single user, then some degree of inconsistency may be acceptable between the tool repository and CM workspace. If, however, multiple users share the same CM workspace, then consistency becomes more of an issue.

Figure 2-1 illustrates the above example in more concrete terms as a lattice of factors contributing to a decision on how to integrate a tool and CM system. In Figure 2-1, orthogonal factors arising from software process, services and mechanisms are combined. A precise definition of workspace semantics is not necessary to understand this illustration. In Figure 2-1, integration of a single-user SMARTSystem[5] workspace with a multi-user NSE[6] workspace suggests a scheme whereby objects are locked in the NSE workspace prior to being imported by

---

5. *SMARTSystem*—a C programming tool marketed by ProCASE, 3130 De La Cruz Blvd., Santa Clara, CA. All future references to SMARTSystem (or “SMART”) refer to version 1.6 of this product.

6. *NSE*—the Sun Networked Software Environment. Sun Microsystems has informally announced that in the near-term NSE will no longer be a Sun supported product.

SMART. Conversely, integration of a multi-user SMART workspace with a single-user NSE workspace suggests that CM can occur within SMART without the need to inform NSE. The

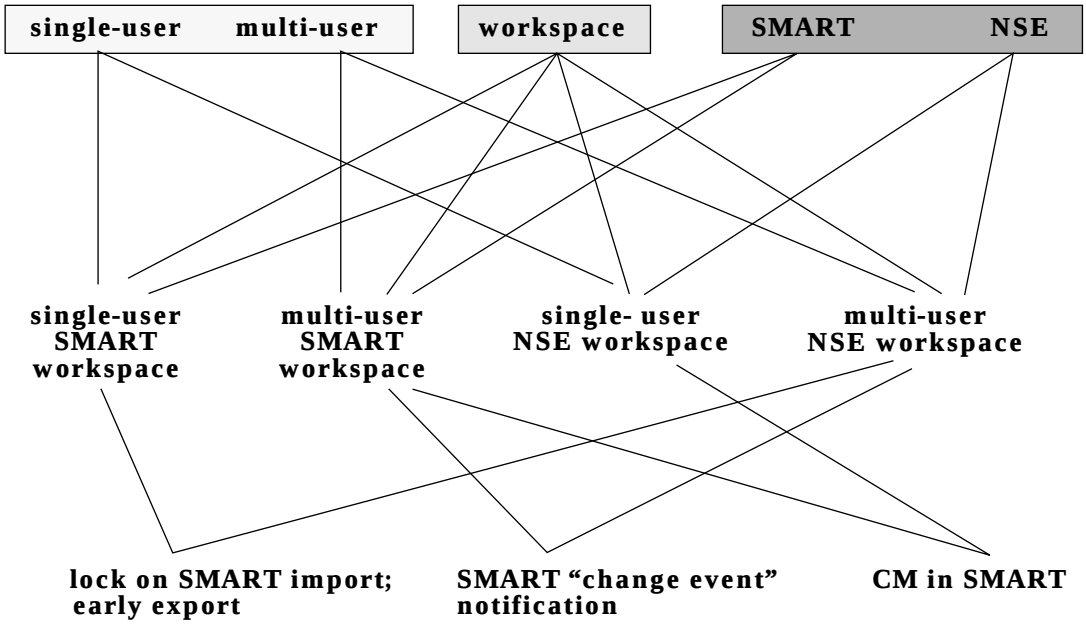

Figure 2-1 Simple Lattice of Factors of Integration Design

example in Figure 2-1 is drawn from an experimental integration of a programming tool (SMARTSystem) and a CM system (NSE). More details on this experiment are available in Appendix A of this report.

Thus, process, services and mechanisms have complex interrelationships which have an effect on CASE integration. The discussion of key concepts in this attempts to provide some structure to the complex factors[7] which drive CASE/CM integration. These factors are classified according to three levels of integration (as described in [37]): processes, services and mechanisms. While some of the factors discussed are applicable to CASE integration in general, an attempt has been made to identify those factors which are particularly relevant to the problems peculiar to CASE integration with CM.

7. The term "factors" and "concepts" will be used interchangeably in the following discussions.

2.2 Process concepts

There has been a growing awareness of the importance of software engineering process concepts in general [26], and with the association of these concepts with software engineering support technology (a.k.a. software development environments) [28]. Software process concepts are particularly relevant to CM, as CM can reasonably be characterized as a discipline (or *process*) for managing change. Documents such as [18] outline many key process concepts necessary for establishing an effective CM discipline.

For this report, a narrow range of process issues are discussed. First, issues related to CM-specific processes are discussed. Then, more generic issues related to integration of tools and CM to life-cycle processes and individual life-cycle steps are discussed. Finally, aspects of software process that are less well-understood—the effects of process modeling notations and the effect of process evolution on SDE technology—are discussed.

2.2.1 Organizational processes and CM applications

CM support technology (i.e., tools and systems) has evolved to provide direct support for many of the CM processes described in [18]. In addition, recent advances in CM support technology have moved beyond the traditional view of CM as a *control* discipline, and now provide *support* for software development activities [15]. As yet, no single CM system encompasses the spectrum of concepts necessary to support the many different perspectives of, or requirements on, CM support technology [13]. Instead, different tools offer unique blends of CM functionality, with different emphasis on different CM requirements. As a result, it is likely that within any organization several CM systems may need to co-exist; further, each of these CM systems may imply constraints on the desired integration of CASE with the CM support services.

One useful way to differentiate the kinds of emphasis provided by CM systems is to classify these systems according to the community the CM system supports. Such a classification, proposed below, can provide a tool integrator with a needed perspective in understanding the relationship between the CASE tool functions and the CM system functions:

- **Corporate CM (CCM)**. CCM looks at software development from the global business perspective. The concerns of CCM frequently center on issues related to profitability—for example, product and market impact analysis for determining product evolution strategies. Corporate CM also provides the context within which CM of a product spectrum can occur. Finally, corporate CM enables data obtained from the management of product spectrums to be applied to a conscientious process improvement program.
- **Project CM (PCM)**. PCM is related to the development of a single product, and in particular affects the control of changes to a product. The traditional CM concepts such as change control board (CCB), access control, and multiple baselines, can be thought of as supporting the PCM perspective. The concept of life-cycle process management is particularly relevant to PCM.

- **Developer CM (DCM)**. DCM supports software developers in making changes to software. There are many aspects of such support, including automated change propagation, smart re-compilation, CM management of developer workspaces, and long transactions. One of the recent advances in CM that has been observed is increased support for software development activities, as opposed to the control-oriented disciplines of PCM. Feiler describes these advances in terms of four distinct paradigms of CM [15].

As indicated above, it is likely that no single CM system will provide support for the broad spectrum of requirements implicit in the three classes of CM applications outlined above. Thus, CASE integration with CM may need to address not just integration of tools with CM services, but with a family of CM services, perhaps provided by different CM systems. A separate, but related, concern is the integration of various CM systems with each other, e.g., metrics gathered by project CM services made available to corporate CM services.

2.2.2 CM processes and the software life cycle

One popular way to describe software process is by way of a life-cycle model. Descriptions of software life cycles abound [4], but all share some common concepts—specifically, the idea of life cycle phases, and the progression of the process through these phases. Alternative models differ about whether the life cycle is iterative, whether several phases can be active simultaneously, etc. For CASE/CM, there is a strong connection between the life-cycle process and the appropriate integration solution.

We have found that it is useful to consider the relationship between CASE/CM and the software life cycle in terms of at least two kinds of relationships: CASE/CM to the process as a whole, and CASE/CM to activities which occur within the context of specific life-cycle phases. For rhetorical purposes we refer to these separate relationships as *life-cycle integration*, and *life-cycle step integration*, respectively. Life-cycle integration addresses global process constraints, while life cycle-step integration addresses the more interactive tool-level aspects of a software development environment.

2.2.2.1 Life-cycle integration

In many ways the metaphor for large-scale integrated project support environments (IPSE) revolves around the notion of control and management of the software life-cycle. Environments such as the Software Life-Cycle Support Environment (SLCSE) [32] typify environment architectures designed to explicitly address life-cycle issues. For CASE/CM integration with the software life-cycle, two kinds of issues arise which are of particular significance: the generation and maintenance of documentation produced as a consequence of life-cycle requirements, and the coordination of tools spanning different activities (life-cycle steps) of the life cycle.

Life-cycle documentation

As many developers of large-scale US DoD software systems can testify, life-cycle standards such as Mil-STD 2167 require the production and maintenance of various kinds of documentation as a means of controlling, and managing, the software life cycle. Some kinds of docu-

mentation directly effect the kinds of CM services used (and, perhaps, how the services are used)—for example, requirements for maintaining *unit development folders* to document every change to *software configuration items*. Other kinds of documentation may also be required which reflect traceability across various phases of the life-cycle, e.g., tracing requirements to test cases.

The issue of traceability is particularly troublesome in current-generation CASE-tool environments because of tool architectural characteristics which make data sharing among tools difficult. Obvious problems such as tool use of private, tool-specific repositories are amplified by the different data models, data formats and non-standard semantics in use among tools provided by different tool vendors. Environments such as SLCSE and Boeing's Advanced Software Environment [24] attempt to overcome these difficulties by mapping tool-specific data to a global data model representing life-cycle artifacts. Other approaches, as reflected in systems such as IBM's AD/Cycle [23], attempt to bypass the need for external agents to perform tool-to-repository data mapping by specifying a standard data model to which vendor tools are intended to comply *a priori*.

The SLCSE and AD/Cycle approach clearly involve great expense and effort; some of the issues raised by these approaches are discussed in [10]. For the purposes of practical third-party CASE integration, a more reasonable near-term approach is to understand and document the life-cycle data required, the relationships between this data and various CASE tools, and procedures (manual, semi-automatic and automatic) for generating this data.

Tool coordination across life-cycle steps

Closely related to the question of traceability is the coordination of tools across different life-cycle steps. Such coordination affects CASE/CM integration because:

- One facet of CM is change control, i.e., controlling *what* changes can be made, *why* they should be made*, when* the changes should be made and *who* should make the changes.
- Tools featuring private repositories need to work within a global CM control regime for effective global change control to be put into practice, yet tools frequently provide their own services and mechanisms for controlling change.

The tool coordination alluded to above will be difficult to achieve given the difficulty of achieving even rudimentary traceability in a CASE environment. However, some coordination of CASE tools with CM in the life-cycle context can be achieved if the problem is viewed in a kind of data flow framework. For illustrative purposes, assume an environment is supported by several distinct CASE tools, each specialized to a specific life-cycle phase (e.g., requirements tool, design tool, coding tool, testing tool), and each of which has its own tool-specific repository services. This is justifiable since many CASE tools are closely related to activities performed in specific life-cycle steps.

Figure 2-2 illustrates some of the CASE/CM integration issues raised by different data flow models of the life cycle. In Figure 2-2, only a subset of life-cycle steps is illustrated, and their

relationships to each other spans overly-simplistic sequential models (i.e., a pure “waterfall”), to more complex (and realistic) scenarios. Also, only a limited set of the many possible design issues that relate to the software life cycle is illustrated.

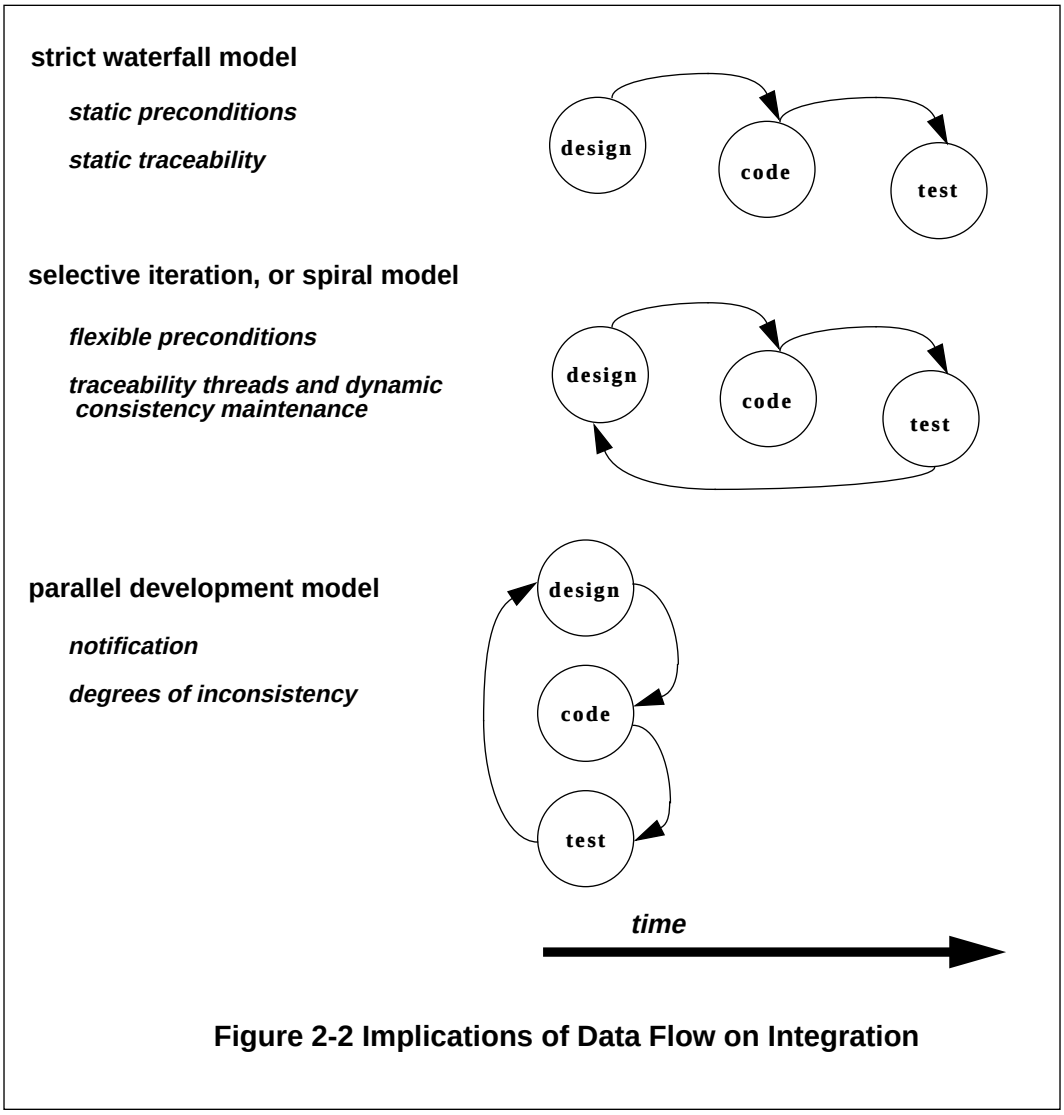

Figure 2-2 Implications of Data Flow on Integration

For the (simplified) waterfall model, two issues are static pre-conditions and static traceability. Preconditions refer to the constraints imposed by the life-cycle model on the circumstances in which the next life-cycle phase (and hence the next tool, in this discussion) can be invoked; static traceability refers to the links created among the artifacts produced by tools in each life-cycle phase. Obviously the lack of iteration implies that once established, at least the source of the traceability link can be made immutable. The introduction of iteration requires the pre-conditions to be loosened to accommodate iteration. Similarly, traceability support must be enhanced to address versions of traceability relations, or traceability threads resulting in DAGs through versions of artifacts. Finally, the introduction of parallelism in each of the life-cycle

phases implies degrees of incompleteness in each phase (why else would there be parallel activity?), as well as the possible need to notify tools and users where logically shared (or derived) information is under modification.

The above examples serve to illustrate the kinds of issues related to the software life cycle that have an effect on the integration of tools, and with CM services. Even the simplistic assumptions about the relationships between tools and life-cycle phases did not shield us from difficult-to-enumerate interactions between CASE integration and software process. If we remove these simplistic assumptions about tools and life-cycle phases, as illustrated in Figure 2-3, the issues become still more complex.

In Figure 2-3, several tools are depicted which could reasonably be available in a CASE environment.[8] The important point to note is the overlap among various *kinds of* tools. In some cases the overlap is intentional and benign (if not beneficial). For example, a design tool such as *StP* can be thought to overlap coding tools to the extent to which StP can be used to generate module interface specifications. In other cases, tools can overlap but provide services which amplify the effect of the tools. For example, the *gIBIS* design journal can help capture alternative designs and design rationale that are not effectively captured in StP. In still other cases, the overlap can be useless (if not destructive). For example, it is not at all clear that there are any benefits to be gained from using two competing design notations and tools simultaneously (e.g., *AGE* and *StP*). Note that Figure 2-3 does not include references to CM tools, which could span all life-cycle phases and overlap with services provided by several of the tools depicted in Figure 2-3; this also provides fertile ground for mutually beneficial and mutually destructive interactions among tool and CM services.

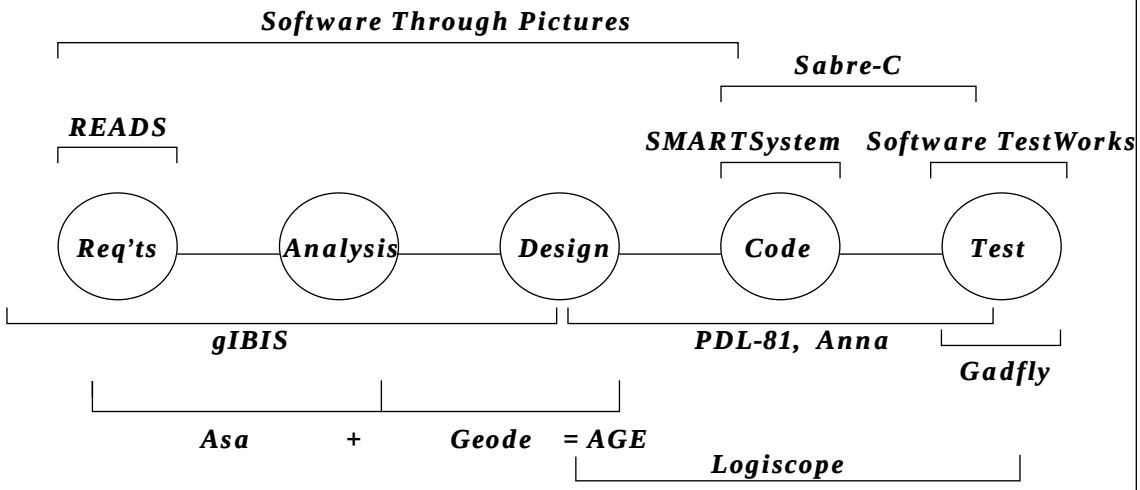

Figure 2-3 Tools and Life-Cycle Overlap

2.2.2.2 Life-cycle step integration

By its nature, processes within life-cycle steps are likely to be more interactive, more collaborative, and more complex in terms of number, kind and frequency of process-related events and data. Further, support for activities within life-cycle steps falls into the realm of the CASE tools; thus, many of the issues related to life-cycle step integration are highly dependent upon tool idiosyncracies. This point about tool idiosyncrasies deserves some elaboration.

Tools are not developed by vendors without some pre-conceived notions of software process. In most cases, there is no strict separation of tool functions and process constraints that can, and should, be separate from the tool functions. For example, it is not uncommon for CASE tools to provide rudimentary CM services to support multiple users. These tool-specific CM services imply constraints on how group members can coordinate; worse, these tool-specific constraints may conflict with the constraints imposed by other tools, or with the broader software processes specified by an organization intent upon a conscientious process improvement program.

One way to express these difficulties is to view each tool, or system, simultaneously from the three levels of integration (process, service and mechanism). Figure 2-4 illustrates this view by showing peer-level integration among the different levels of NSE (a CM system) and SMARTSystem (a C programming development tool). Figure 2-4 views the integration of each system in depth, and illustrates well how peculiarities of each system at each level of integration must be reconciled in order to achieve satisfactory integration.[9]

2.2.3 The impact of process modeling notations

The previous discussion on life-cycle models raises the question about how to best describe, or model, processes. These discussions were not meant to suggest that a structured analysis approach to data modeling was desirable, but merely to illustrate the relationships between global software processes and CASE/CM integration. Although the recent popularity of software process has resulted in a diversity of approaches to describing, or modeling, software processes, there is as yet little consensus on what modeling formalisms are best suited for what purpose and for what audience. For CASE integration, a modeling emphasis on data objects (as might occur if process models were described in PCTE schemas) may result in an emphasis on data integration, while the use of behavioral models or state/transition models may result in a greater emphasis on control integration.

The relationships between software processes, process models and modeling formalisms, and environment adaptability to software processes is an area of active research [28].

---

[8] Not all of these tools are commercially available. Further, not all are logically compatible, e.g., Anna is an Ada annotation language, while Sabre-C is clearly a C development tool; however, given the prospect of multi-lingual development efforts, even in this case some form of integration may be reasonable.

[9] Note that for SMARTSystem, only those services related to CM, i.e., potentially in conflict with NSE services, are illustrated.

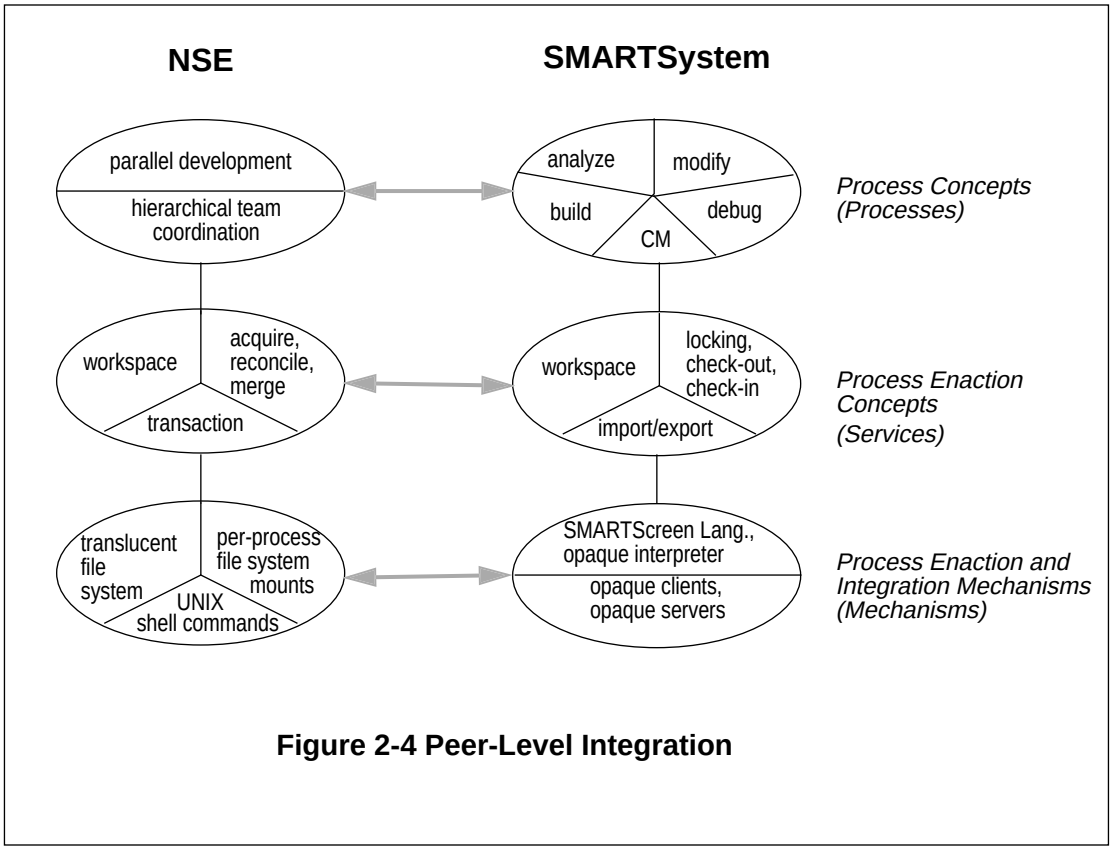

Figure 2-4 Peer-Level Integration

2.2.4 Process evolution

Since software processes have an effect upon the design of CASE/CM integration, the possibility that changes to the process (through evolution and enhancement) may invalidate integration design decisions should be considered. Thus, the desire to support particular process constraints directly in the integration of tool mechanisms must be weighed against the cost of automating such a process constraint and against the likelihood that this constraint will change.

2.3 Service concepts

The notion of services has emerged as an important component of our evolving understanding of integration in a software environment ([30], [29] and [1]). While no precise definition of the term is provided in any of these sources, the idea in general is that a service is an abstraction of a function provided by a software environment. The nature of the abstraction is that it conceals the details of who provides a function (i.e., a tool or framework) and how it is provided (i.e., its syntactic interface).

For an integrator, the service concept is useful because it provides a layer of separation between vendor-specific implementation idiosyncracies and the objective requirements of integration, as defined by the software process. This separation helps the integrator focus on what

the integration is intended to accomplish, not on how to accomplish it. Further, services help the integrator identify commonality and variance among the tools being integrated (see discussion of services profiling, below).

Much more work needs to be done to flesh out the details of services, especially their implication on software environment architectures. The following sections discuss the meaning of services, particularly with respect to CM (service domains); how to make use of services in attacking an integration problem (services profiling); and longer-range issues of services and service standards (service evolution).

2.3.1 Service domains and primitive and abstract services

Services are abstractions of functions provided in a software environment. Service *domains* are a further abstraction which group logically related services into coherent collections. A reasonable analogy would be to equate a service domain with abstract data types or abstract machines whose purpose is to support, or enact, software processes. This view of services provides a richer conceptual model to the integrator than the arbitrary partitioning of environment functions into framework and tool services, or the somewhat shallow distinction of horizontal versus vertical functions. Service domains are useful for several purposes. They:

- Provide a basis for expressing relationships between services.
- Provide a mapping between groups of services and higher-level activities in the software process (e.g., “design”).
- Provide a mapping between groups of services and the underlying support technology (e.g., CASE tools) which “package” these services.

Reference models such as [29] and [1] are a start at partitioning services into services domains. In the commercial world, *CASE Communiqué* [8] is attempting to achieve CASE vendor consensus on the protocols necessary to integrate tools into Hewlett-Packard’s SoftBench environment. These protocols constitute an implementation of services, with SoftBench protocol *classes* corresponding to service domains. Thus, while the notion of service is still relatively new and unstable, progress is being made in understanding, defining and standardizing services. The CASE Communiqué activity is of particular interest to tool integrators.

An understanding of the essential elements of CM services is mature in comparison with that of other service domains, largely because CM is a common discipline across all software development projects, and because of the wide variety of research and commercial CM products available to support the CM discipline. The work of Katz [19], Feiler [15] and Dart [13] is representative of the state of maturity of CM concepts. Thus, a tool integrator is in a reasonably strong position to characterize the elements of CM (process, service and mechanism) that will contribute to the integration of CASE with CM.

One interesting result from Feiler and Dart is the emergence of a view of services as interrelated, perhaps multi-level, abstractions. That is, rather than being defined as a flat collection of services, some services may be more “primitive” than others, and may serve as building blocks for higher levels of service abstractions. For example, the transaction *paradigm* dis-

cussed in [15] can be thought of—and implemented in terms of—lower-level CM services, some of which are defined in [13], e.g., workspace and transaction *services*. It is just such an understanding of the deeper structure of services such as CM that will support the development of more effective integration of tools in support of software processes.

2.3.2 Services profiling

The discussion in the previous section focused on describing some of the qualitative aspects of services—what they are and what they are useful for. In this section, the focus shifts to the use of a specific analytical approach for understanding the effect of services on CASE/CM integration design decisions. This analytical approach—services profiling—is described in [30]. In this discussion we make use of profiling to illustrate its use in CASE/CM integration.

2.3.2.1 Profiling with respect to CM application areas

As described in [30], profiling represents a view of services from a particular perspective. One perspective that is useful in CASE/CM integration is to profile levels of support for the three different CM application areas (corporate, project and developer CM) described earlier in this report. Figure 2-5 illustrates one way to view such a profile when applied to two different CM

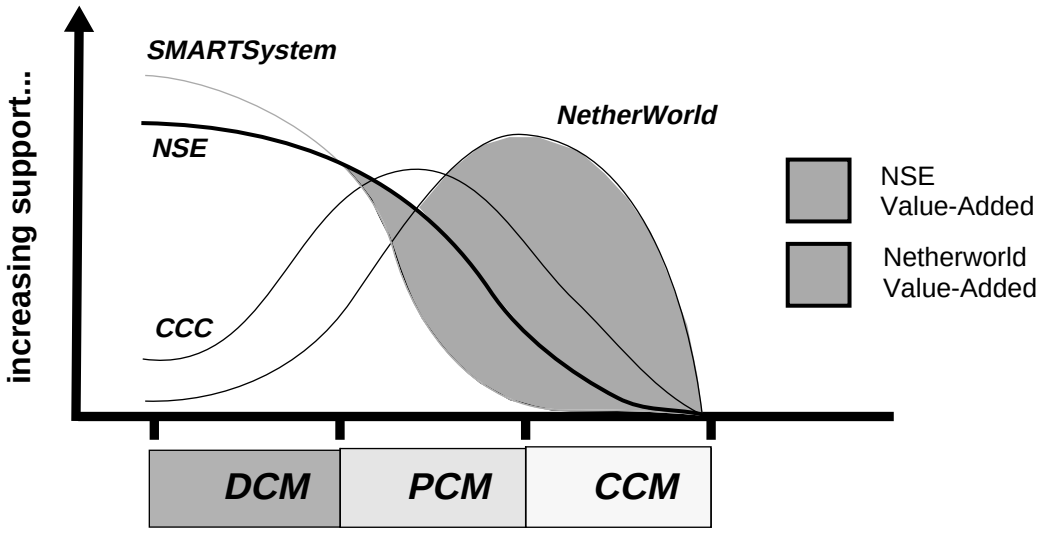

Figure 2-5 Profiling Systems by CM Application

systems (NSE and Netherworld, an internal National Security Agency CM product) and a CASE programming-support tool with a significant CM component (SMARTSystem). The figure is not meant to convey the results of a concrete analysis, but rather to convey the different foci of CM support offered by different systems, and the potential effect these different foci have on selecting and integrating systems. In Figure 2-5, both SMARTSystem and NSE show

similar profiles for developer, project and corporate CM. A reasonable question to ask is whether there would be a practical cost/benefit ratio to justify an integration effort. Conversely, functions provided by NetherWorld complement those provided by SMARTSystem, so intuitively there would be ample scope for enhancing an environment composed of these systems by integrating them.

Unfortunately, such an analysis is too simplistic. Besides being unsatisfactory on a strictly quantifiable basis, the cost/benefit analysis alluded to above must consider the semantic (services) and mechanistic compatibility of SMARTSystem with NSE versus SMARTSystem with NetherWorld. In Figure 2-5, for example, one could also conclude that NSE services harmonize better with SMARTSystem than do NetherWorld's. As such, both the cost and effect of a SMARTSystem integration with NSE might be more satisfying—and practical—than an integration of SMARTSystem with NetherWorld. Thus, while such an intuitive profiling technique might be useful for coarse "anatomical" comparisons of tools, a comparison at a finer level of services granularity is necessary

2.3.2.2 Expanding on CM applications

We can make use of Dart's work as a basis for identifying CM services, and use these services as a basis for characterizing the CM applications. In Figure 2-6, a subset of the CM requirements discussed in [13] that may be considered as supportive of developer CM defines a perspective for profiling NSE and the CM component of SMARTSystem. This profile depicts the level of support for various developer CM services on a scale of *no* support, *medium* support and *high* support (with the circumference defining high support). The three-tiered ranking is

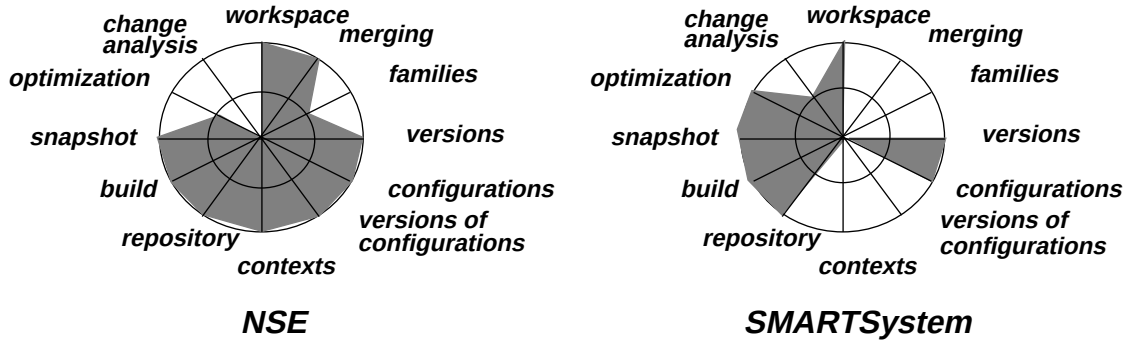

Figure 2-6 Profile of SMART and NSE by Developer CM Perspective

useful not only for identifying areas of overlapping support, but also in (loosely) quantifying a level of support that may be useful in later determining how to allocate responsibility for services across tools. For example, SMARTSystem has stronger support for build optimization than does NSE. This might indicate that SMARTSystem should manage the build-related ac-

tivities of developer CM; conversely, NSE would be in a position to describe versions of configurations, which is something SMARTSystem does not support at all.

Given a detailed profile as shown in Figure 2-6, a more detailed analysis of overlapping services can be performed. Such an analysis is necessary because:

- the services are too broad (as currently defined) to serve as a basis for making detailed integration design decisions.
- the services are not standardized, and so different vendor implementations may impose different semantics on the services.

As an example of the second point, both SMARTSystem and NSE support workspace services and transaction services (not shown in Figure 2-6). Table 2-1 illustrates this point. SMART-

SMARTSystem	***NSE***
Transaction	**Transaction**
• **1-level transactions**	• **nested transactions**
• **pessimistic concurrency**	• **optimistic concurrency**
• **total commit (no local memory)**	• **partial commit (local memory)**
Workspace	**Workspace**
• **single user**	• **multiple users**

Table 2-1 Implementation Differences in Services Semantics

System transactions and workspaces are shown to be more restrictive than their NSE counterparts. However, the use of SMARTSystem transactions and workspaces is intrinsic to the tool, and hence can not be delegated. The job of the integrator, then, is to reconcile the differences between SMARTSystem and NSE services in order to make more effective use of the strengths of each service. A further elaboration of SMARTSystem/NSE details can be found in Appendix A of this report.

2.3.3 Services evolution

As already mentioned, industry consortiums such as CASE Communiqué are beginning to address *de facto* standardization of services and interfaces to services. At a more conceptual level, project support environment reference models are making use of the concept of service to describe and compare environments [1] [2] . Finally, work is in progress within the SEI Software Development Environments project to extend and enrich the CM concepts found in [13]

with a more complete and formal model of CM services, and make use of these services in CASE/CM integration.

2.4 Mechanism concepts

In the preceding discussion of services, the notion of services profiling was described as ultimately resulting in a comparative view of two (or more) systems. The objective of such comparisons was to expose commonality and variance among the services provided, and the semantics associated with these services. In Table 2-1, the differences between the transaction services offered by SMARTSystem and NSE could be attributed to implementation, or mechanism-level, details. These differences emerge as a result of a lack of standards for services, and the (still) shallow semantics associated with services as they are emerging from organizations such as the CASE Communiqué.

However, there are other mechanism-level factors that have a significant effect on the design of CASE/CM integration solutions. From our experiments in CASE/CM integration, we have found it useful to think in terms of three major categories of mechanism factors that contribute to an integration design: tool architectures, CM architectures and CASE/CM integration architectures. These categories are discussed in more detail below.

2.4.1 Tool architectures

The idea of describing characteristics of tools that make them integrable (or un-integrable) has been explored by Nejmeh in [25]. Nejmeh's approach is to describe these characteristics as attributes of the tool as a whole, e.g., use of standards such as window system and licensing schemes, and sensitivity to environment characteristics such as location independence of tool binaries and data.

We have found it useful to consider two additional dimensions of tool characteristics—its architecture with respect to data management and its low-level process structure— when thinking about CASE/CM integration.

2.4.1.1 Data management architectures

It is not surprising that the data management architecture of tools should be a significant factor in integrating the tool with CM services. Where a tool stores data (if at all), how it accesses data, and what kinds of data it manages are all likely to impinge on the practicality of various CASE/CM solutions.

We have identified four broad classes of data management architectures, each of which implies its own set of integration issues:

1. **Filter tools**. As the name implies, this class of tool follows the UNIX model of a tool which processes data independently of where the data is located. For example, the UNIX tool *grep* can process data residing in files, or data received from *stdin*; likewise its results can be placed in a file, or passed along on *stdout* for use by other filters.

2. **Deriver tools**. This class of tool performs transformations on various types of files. In the UNIX tradition, file types are distinguished by naming conventions, frequently involving the use of file suffixes. For example, the UNIX C compiler cc transforms C text *source* files (files with ".c" suffixes) into re-locatable binary *target* files (files with ".o" suffixes).

3. **Data dictionary tools.** Like derivers, this class of tool performs transformations on various types of files. Unlike derivers, data dictionary tools typically produce more highly structured and interrelated target files; this additional structure and complexity requires the data dictionary tool to provide data management and data manipulation services to the data dictionary. An example of data dictionary tools includes most commercial Ada compilation systems and design tools such as IDE's Software Through Pictures.

4. **Database tools**. Database tools are like data dictionary tools in that they provide tool-specific data management services. However, database tools also provide data management for the source files involved in tool data transformations. That is, while data dictionary tools may be independent of configuration management of source files, such services are likely to be an intrinsic part of a database tool. An example of a database tool is ProCASE's SMARTSystem.

Effect of data architectures on CASE/CM issues

Filter tools are relatively uninteresting from a CM integration perspective. The only issue that arises concerns the order in which filters are applied to transform a source object to a target object (to re-establish consistency); but in these cases the order of application is usually "fixed" in a special wrapper program, and in any event these situations can be thought of as a special case of deriver tool application.

Deriver tools form the bulk of conventional software development tools (compilation, linking, text processing, document generation, etc.). The key issues related to deriver tools tend to focus on maintaining consistency of target files with source files; the UNIX *make* program is an old favorite here (though it is showing signs of age).

The opaqueness of the structure of derived objects and their relationships to source objects in data dictionary tools introduces several complex problems. For example, the effectiveness of CM-provided build services can be greatly diminished if these services can not recognize build inconsistencies, or perform any optimization in re-establishing consistency. For large data dictionaries this can be a serious problem.

Database tools introduce a slightly different set of issues than data dictionary tools. In addition to the likelihood of completely invalidating CM-provided build services, an additional problem of source-data duplication arises. That is, while derived data can be re-generated (and hence inconsistencies can be resolved through a build process), maintaining duplicate copies of source data in a CM and tool repository is a more serious cause for concern.

Some of these issues will be discussed again in the abstract illustrations of CASE/CM integration issues in Section 3 of this report.

2.4.1.2 Operating system-level process architectures

The term “operating system (OS)-level process architecture” refers to the operating-system level processes used by the tool to carry out its functions; this is meant to distinguish these process characteristics from the software-engineering processes discussed earlier in this report.

OS-level processes are typically not among the overt characteristics of tools, yet the OS-process architecture of tools can play a significant role in establishing and maintaining communication among different tools. While tools can be constructed which make use of an arbitrary number of processes connected in arbitrary ways, we have identified four main kinds of OS-level process architectures *that can affect CASE integration with CM*. These architectures, illustrated in Figure 2-7, capture two orthogonal and interacting features: process *life-span* and

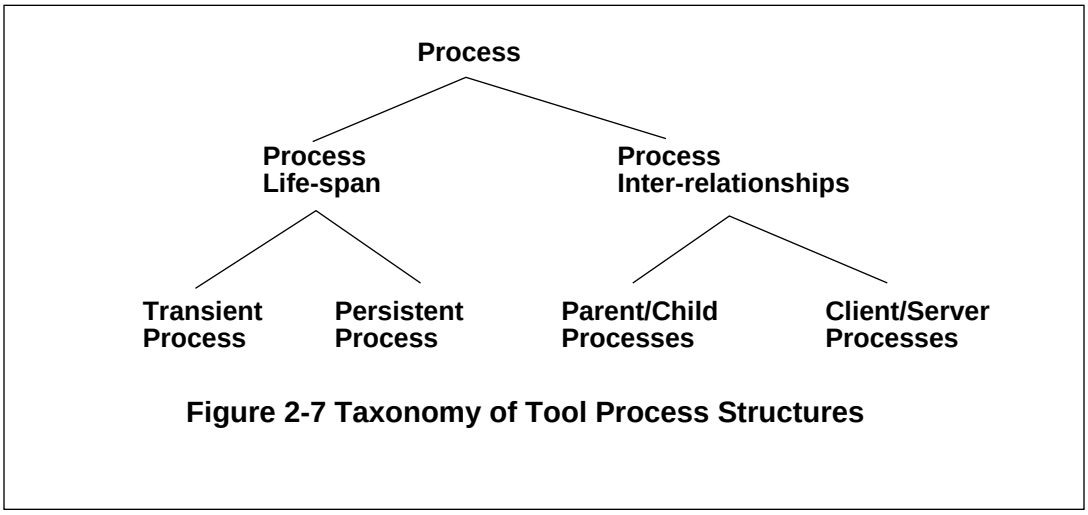

Figure 2-7 Taxonomy of Tool Process Structures

process *inter-relationships*. These architectures, and the meaning of the orthogonal classification depicted in Figure 2-7, are discussed below.

1. **Transient tools**. This class of tool follows the traditional UNIX tool model, i.e., a tool is launched from the command line, executes until it completes a discrete task, and then terminates. Although transient tools may make use of several processes, perhaps related in parent/child or client/server fashion, the important distinction is that the duration of tool execution is tied to the completion of a discrete task. Examples of transient tools include most common UNIX utilities, e.g, *grep, cc, make*.

2. **Persistent tools**. Unlike transient tools, persistent tools may be thought of as providing tool services which may be used repeatedly by tool clients to perform certain kinds of tasks. Whereas transient tools “terminate” after completing their task, persistent tools remain active servers. The advent of OS support for lightweight processes and multiple threads of execution may make persistent tools more common. An example of a persistent tool is the text editor included in the Hewlett-Packard SoftBench environment.

3. **Parent/child tools**. This class of tools is similar to persistent tools, except that the parent tool is persistent but that it “forks” or “spawns” transient tools that are tied to particular activities. The subtle difference between persistent and parent/child tools is that child tools have independent process contexts from parent tool process(es); however, in most cases the child process assumes the process context of the parent context. The GNU Emacs editor has features which supports creation of child editors.

4. **Client/server tools**. This class of tools has properties of both persistent and parent/child tools. The server acts as a persistent tool, while the client is typically a transient tool which makes use of the services provided by the server. Unlike parent/child tools, the client process need not have any process context in common with the server process. This subtle difference is very significant. PROCase SMARTSystem is an example of a client/server tool.

It should be noticed that not all tools fall neatly into one of these categories. In particular, tools may exhibit characteristics of several (or all four!) process architecture classes. For example, Frame Technology’s FrameMaker is a client/server tool because it provides server functions for FrameMaker clients; it is a parent/child tool since a top-level control panel is used to “fork” child FrameMaker sessions; the “forked” FrameMaker sessions are persistent tools since they may be reused to create and modify any number of FrameMaker documents. Despite this, however, these process architecture classes do provide some insight into the kinds of issues that will arise when integrating CASE tools with CM.

Effect of OS-level process architectures on CASE/CM issues

The distinction between process life-spans and process interrelationships is motivated by the advent of network file systems (and like services) which require, or are implemented, through the use of per-process context information. Process life-spans are related to how per-process contexts are established and maintained; process interrelationships are related to how processes may share the same, or communicate across different, process contexts.

For example, in UNIX environments running the Sun Networked File System (NFS), different host machines may have different file systems “mounted” and thus two tools communicating across different network nodes may have different images of the file system. Transient tools acquire their view of the mounted file systems at the time they are invoked and maintain this view during their entire life-span. Since persistent tools typically do not migrate across different host machines, re-establishing a new process context appropriate for different per-host file system mounts is usually not an issue; however, re-establishing a new process context could be important for persistent tools where advanced CM systems are used (see discussion, below). Continuing this NFS example, multi-process tools may result in, for example, servers running on one host machine while clients run on another. In this scenario, ensuring that both client and server are viewing the same file system is problematic, since different file systems may be mounted on different machines under the same name. These examples illustrate the problems of establishing per-process contexts, and in sharing per-process contexts across multiple processes.

Such problems also arise from advanced CM systems that offer pervasive versioning as part of the underlying operating system file system. Examples of such pervasive version management systems include the Apollo Domain Software Engineering Environment (DSEE) [22], NSE [33] and more recently, the AT&T 3-D File System [21]. In these systems, processes are started with respect to a view of the file system that corresponds to a desired view into the version space of files maintained by the file system. Thus, in order for client processes to communicate meaningfully with server processes (where communication involves identification of versioned files), both processes must share the same process context.

In these situations it is often necessary for the tool integrator to understand some aspects of the tool architectures which might not be documented or readily apparent. For example, parent/child tools may have no difficulties with per-process context since the UNIX *fork* primitive causes process contexts to be duplicated from parent to child process; however, some operating systems (including newer flavors of UNIX) provide more flexibility in interpreting the semantics of UNIX *fork*. Should the above systems' model of providing pervasive version management become widespread, it might be reasonable for tool developers to include services that allow persistent tools (and, by extension, servers) to change their process context. This may also require the acquiescence of operating system vendors.

2.4.2 CM architectures

The previous discussion on tool OS process-level architectures touched on the importance of the architecture (or implementation) of the CM system in designing a CASE/CM integration solution. Two classes of CM architecture are obvious, and imply different CASE/CM integration issues:

1. **CM tools**. This is perhaps the most "traditional" way in which CM services are provided. CM tools are separate, stand-alone programs that are executed explicitly by some agent (a user or another computer program). It is reasonable to view CASE tool integration with CM tools as a special case of tool-to-tool integration [34].

2. **CM systems.** Systems include the examples cited in the previous section—DSEE, NSE, AT&T 3-D File System. Also included are *integration frameworks* which provide version and configuration management services as part of the framework itself. Examples include the PACT [36] version management services which are available in at least one commercial implementation of PCTE 1.5, and are included as part of the ECMA PCTE standard [27], and integration framework specifications such as the CASE Integration Standard (CIS) [9]. It is reasonable to view CASE tool integration with CM systems as a special case of tool-to-framework integration [34].

The significance of integrating tools with CM tools versus CM systems hinges upon the balance between the benefits (e.g., simplicity) of integration provided by pervasive, transparent VM and CM services weighed against the possibly subtle implementation dependencies (such as per-process context) in the way these services are provided. The integration discussed in Appendix A highlights some of these balances.

2.4.3 CASE/CM integration architectures

The previous sections have discussed the effect of tool and CM architectural characteristics on the design of a CASE/CM integration. It is also useful to think of architectures which ex-

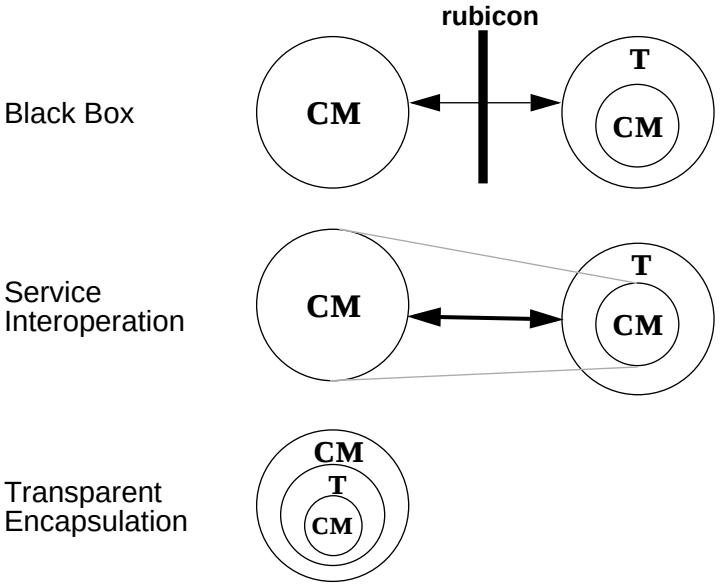

Figure 2-8 Integration Architectures for CASE/CM

press characteristics of the combination, or integration, of CASE with CM (tools and systems). Three kinds of CASE/CM architectures are distinguishable, as illustrated in Figure 2-8, and raise different design issues:

1. **Black-box integration**. In many ways this is the most straightforward and most common form of CASE/CM integration. The essential idea is to treat the tool as a black box, into which data is imported and from which data is exported. The CM system is not aware of the details of the activities or structure of tool data; neither is the tool aware of CM details. In Figure 2-8 this is illustrated by the thick “rubicon” which separates the tool from the CM system.

2. **Service integration and interoperation**. This form of CASE/CM integration implies a more fine-grained interaction among the tools and the CM services. For example, allowing the tool to maintain versioning information such as “change reasons” for fine-grained objects while delegating more global change management information, such as release information, to the CM system. In Figure 2-8 this is illustrated by the relationships (illustrated by the dashed lines) among the CM services provided by the tool and dedicated CM system.

3. **Transparent integration**. Both black-box and service integration requires the intervention of some agent (either a human or a computer program) to manage the dialogue between tool and CM system, e.g., when to export data.

The idea of transparent encapsulation is to make the tool/CM dialogue a more intrinsic part of the tool/CM relationship. One way to achieve transparent encapsulation is to use integration frameworks or operating systems which provide pervasive VM/CM services (see Section 2.3.2, above). Another way is to extend either the tool or the CM system to incorporate and automate the dialogue between tool and CM system. In Figure 2-8 transparent encapsulation is illustrated by nesting a tool within an enclosing CM system.

It is useful to distinguish between end-user transparency and implementation transparency, analogously to the distinctions between well integrated and easily integrated.

As with some of the other mechanism factors discussed above, there are no straightforward partitions among these three classes of CASE/CM integration architectures. For example, a fully-automated black-box integration would appear to satisfy the description of transparent encapsulation. However, the intent of transparent encapsulation is to make full use of both tool and CM services in such a way as not to require manual intervention, so the distinction between fully-automated black-box integration and transparent integration still holds.

2.4.4 Mechanism evolution

One problem confronting tool integrators (and end-users, for that matter) is the constant and rapid pace of evolution in software products of all types. Operating systems, integration frameworks and tools all evolve through new releases, new products, new combinations of products, etc. This rapid evolution is one motivation for viewing integration in terms of a relationship between processes and logical services. Unfortunately, as this section on mechanisms reveals, various implementation factors present in tools and CM systems can have a significant effect on the design of an integration solution. This argues strongly for an approach to CASE integration with CM that does not emphasize aspects of automation that are heavily dependent upon software mechanisms not under the integrator's control. In fact, as Section 3 illustrates, many of the design decisions concerning CASE/CM integration occur at the level of tool usage conventions and frequently do not require much, if any, construction of programmatic (i.e., automated) integration devices.

2.5 Summary of key concepts of CASE/CM integration

Integration of tools in general involves: *process* factors (how a tool is used), *service* factors (what the tool is doing), and *mechanism* factors (how the tool does it). Although existing CASE technology does not feature a strong separation of processes, services and mechanisms, this three-level perspective on integration provides insight into designing an effective integration. An important point to note is that we view the goal of integration as a means of making more effective use of a collection of tools to achieve an objective, and not simply as a means of automating mundane tasks. Thus, the services layer is a useful means of abstracting away from mechanism-level details that dominate much of the commercial CASE integration marketplace, and focusing attention on the more important integration relationship that exists be-

tween the services a tool provides, and how these services should be used to support software engineering processes.

3 CASE/CM Integration Illustrated

The previous section described concepts of CASE/CM integration. In this section these concepts are illustrated through a series of integration scenarios. Since there are three dimensions which can vary in these scenarios (process, service and mechanism), and since within each dimension many possible combinations of concepts discussed in Section 2 are possible, it is necessary to “fix” certain dimensions while allowing others to vary.

In the following discussion, we limit the services involved in integration to simple CM services such as check-out/check-in, workspaces, and locking, and limit the mechanism dimension to deriver and data dictionary tools.[10] The discussion focuses on the implications on software process raised by integrating these tool architectures and simple CM services. First, simple scenarios are described, and complexity is added in succeeding scenarios.

3.1 Deriver tool and check-out/check-in—foundational basis

The base case for the following discussion is a simple deriver tool integrated with a simple check-out/check-in-based CM tool. This scenario is illustrated in Figure 3-1, and introduces

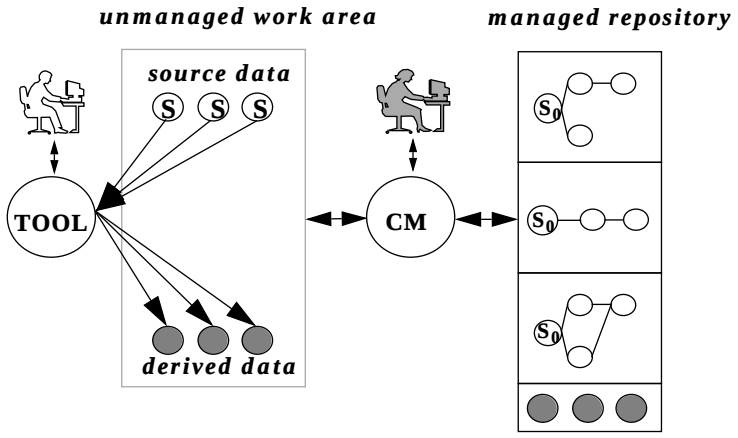

Figure 3-1 Deriver and Check-Out/Check-In

some of the terminology used in the succeeding examples:

- **Source data**: the input to a derivation process.
- **Derived data**: the output of a derivation process.

[10] Appendix A has a detailed description of an integration experiment involving a database tool and transaction-oriented CM system.

- **Unmanaged work area**: a collection of source and derived objects under user management, located arbitrarily in the file system or object management system.
- **Managed repository**: a data repository under CM administrative control.

The simplicity of this scenario is deceptive. The issues raised in integrating deriver tools with check-out/check-in CM tools characterize many state-of-the-practice software development environments. Most seasoned project leaders will recognize some (if not all) of the following issues illustrated by in Figure 3-1:

- Data is exported from the CM system into an unmanaged (from the CM perspective) part of the file system, described in Figure 3-1 as an *unmanaged work area*. The lack of CM control over the developer work area means that developer support is limited (substantially) to locking and change synchronization.
- There may need to be different project roles associated with the export and import of data between the repository and the unmanaged work area. Since the work area is unmanaged, this project role may need to a) determine where and to whom the sources were checked-out, and b) perform quality-control functions when objects are checked-in to the repository.
- There is a question about whether derived objects should be managed along with the source in the repository. The answer to this may depend upon the implementation capabilities of the CM tool, such as whether binary data can be managed in the repository, and whether versions of binary data can be managed.
- If several unmanaged work areas are active simultaneously, the CM system can guarantee mutual exclusion on source file edit operations, but can not guarantee consistency of changes made in separate work areas. Such additional semantic constraints, as found in [12] would need to be added as additional CM services. As a consequence, the repository may become unstable during developer check-in phases.

There are, of course, many more issues that can be raised in this simple scenario. Most of these issues still apply where more complex CASE tools are integrated with CM. However, complex CASE tools also introduce new sets of issues.

3.2 Data dictionary tool and check-out/check-in

As discussed in Section 2.3.1.1 of this report, data dictionary tools introduce data management services for highly structured derived data. A number of different issues arise as a result of integrating a data dictionary tool with CM. Several scenarios (versioned dictionary, private dictionary, shared dictionary) are discussed, below, which highlight these issues.

3.2.1 CM-managed data dictionary

One consideration in integrating data dictionary tools with CM is whether the data dictionary is managed by the CM system. This consideration arises because of the potential for great

expense involved in maintaining the consistency of the data dictionary with respect to source objects that are outside of the control of the data dictionary tool.

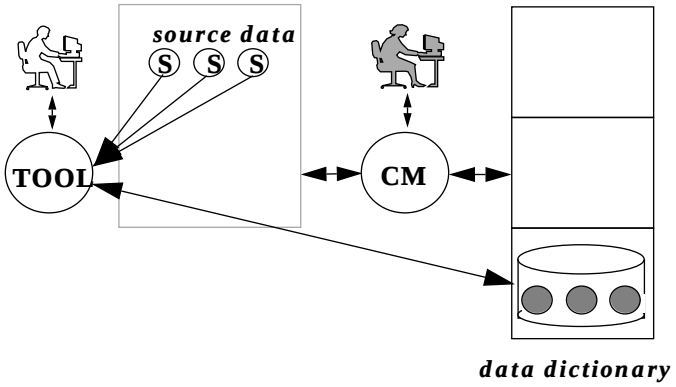

Figure 3-2 CM-Managed Data Dictionary

Figure 3-2 illustrates the CM-managed dictionary by indicating that although source files are manipulated in unmanaged work areas, the consequences of these changes affect the data dictionary within the CM repository. Although this integration scenario has the potential for reducing the costs of maintaining data dictionaries (disk space and re-creation time), a number of troubling issues arise:

- If multiple work areas are active simultaneously, and if one workspace has read-only copies of objects being modified in another workspace (and *vice versa*), the dictionary can “thrash”—i.e., each successive update of the dictionary from one work area will “undo” the changes made from the other work area. The underlying problem is the sharing of derived objects but not the source objects. The result is instability of developer work areas.
- Implementation constraints can render CM management of the repository problematic. On the tool side, it may be difficult to predict or contain the generation of data dictionary objects; such prediction is necessary if the data dictionary is to be versioned properly. Even if versioning is impractical (for example, because the CM repository will not manage binary data), having the CM repository manage an unversioned data dictionary may be impractical if the data dictionary tool creates dictionary objects whose file-system protections will violate repository access control policies.

For reasons such as these, it may be impractical to rely on the CM system to maintain a single copy of a data dictionary. Although it may be advisable to associate a data dictionary with a repository, i.e., as a derived object which reflects the latest (or some other designated) version in the source repository, in practice a closer relationship of dictionaries with developer work areas is required. Three alternatives—private and shared dictionaries, and multiple repositories, are discussed.

3.2.2 Work area-managed data dictionary: private data dictionary

An approach to alleviating one problem raised in Figure 3-2, work area instability, is to associate the data dictionary more closely with the source files. This would produce an analogous integration scheme as illustrated in Figure 3-1—derived files possibly existing in both the work area and in the repository. The scenario in Figure 3-3 goes a little further[11] by illustrating two active workspaces, each with different copies of the data dictionary (hence each stable with respect to derived files).

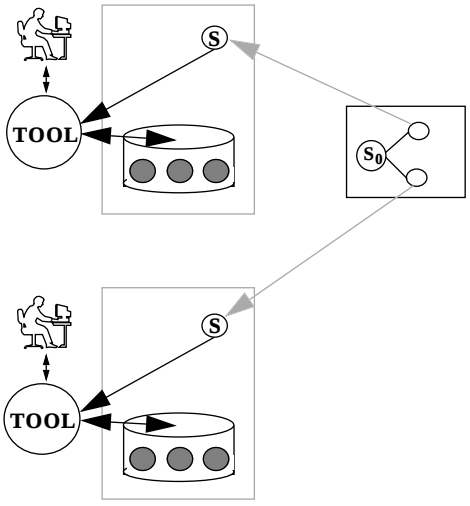

Figure 3-3 Private Data Dictionaries

In this scenario, the repository is used to synchronize access to the shared resources (in this case, the source files) so that the local copies of the derived files (the data dictionary) will be consistent. A number of issues arise with this scenario:

- Copies of the data dictionary can proliferate, introducing significant resource consumption (compute cycles and secondary store).
- The use of locking in the repository guarantees consistency of data dictionaries with respect to source files in the same work area, not with source files in different work areas. Ultimately, a separate merge operation must be performed on the data dictionary, i.e., to combine the source files from the separate workspaces into a single data dictionary. This can be done within the repository or within a designated work area. This extra overhead can be considerable and needs to be weighed against the benefits of work area stability.
- Related to this last point is the interaction of repository locking semantics with the difficulty of performing a data dictionary merge operation. In Figure 3-3,

[11] The extra user role for repository management has been elided from the remaining figures for clarity.

objects on two branches of a version group are being modified. In this scenario, the two source objects will need to be merged; this merge is a separate, but related, merge operation to the merge that needs to be done on the data dictionary. In some situations it may be necessary to merge the data dictionaries as a by-product, or instrument, of merging the source objects. Again, this can be a costly operation. Note that locking the entire version graph for an object may simplify, but does not bypass, the need for a separate data dictionary merge.

In the above scenario, the greater convenience of work area stability is offset by the costs of dictionary maintenance. In the next scenario, the need for dictionary merging is bypassed by having developers share dictionaries.

3.2.3 Work area-managed data dictionary: shared data dictionary

Dictionary sharing is one way to avoid duplication of dictionaries and to enhance coordination of developers. However, the model illustrated in Figure 3-4(a-c) and discussed below requires that some policies be established to coordinate multiple users sharing one derived object. The following scenario leads towards the notion of CM-managed *workspaces* as a means for achieving the same effect.

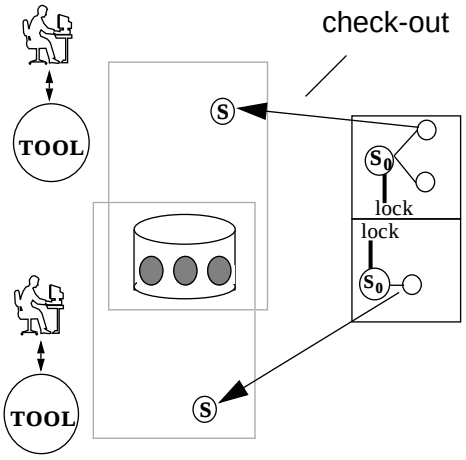

Figure 3-4a Shared Data Dictionaries

Figure 3-4a illustrates the start of the scenario, whereby two developers each manage private copies of source, while sharing a single data dictionary. This is similar to the scenario depicted in Figure 3-2, except that consistency management policies are part of the work area policies managed by the developer (and not the CM system or administrator). Note that this scenario implies that multiple data dictionaries can be created.

In order to avoid the “thrashing” problem described in Figure 3-2, we must arrange that the read-only copy of each developer’s sources are kept consistent with respect to the version of

this source under modification in another work area. Various operating systems and object management systems will have different mechanisms for doing this; in UNIX it can be done through file *links*. The use of UNIX links is illustrated in Figure 3-4b, where the shaded source objects are virtual objects, i.e., are linked to the real source objects.

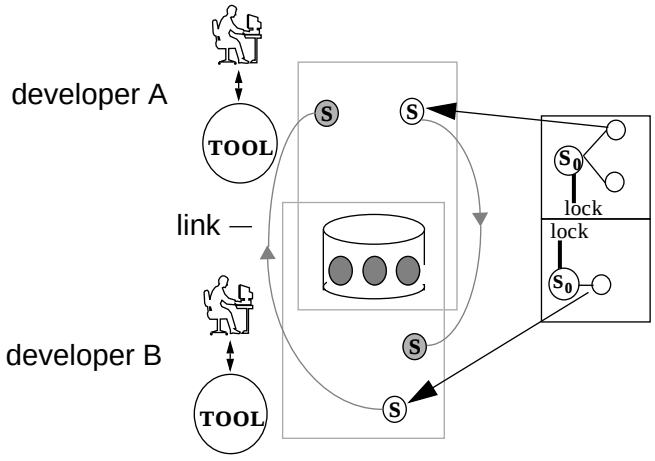

Figure 3-4b Shared Data Dictionaries

The use of links in this way in effect merges the two developer work areas into one logical work area. Clearly, one consequence is that while the dictionary may be guaranteed to be consistent, the work areas as a whole are not stable. Thus, the developers will not have a stable, predictable development environment in which to work.

Further complications can ensue through continued use of these shared work areas, as is illustrated in Figure 3-4c. In Figure 3-4c, Developer B has completed the modifications to the source and has “checked in” the object to the repository. Note the creation of a new version of the object in the repository (highlighted as a blackened circle). At a later date a third developer, Developer C, needs to modify the source recently checked-in by Developer B. Unfortunately, the lack of integration between the CM repository and the informally managed work areas causes a problem. Developer C finds the desired source unlocked, and so copies it to a local work area. Even if Developer C is alert enough to correctly link the needed auxiliary source from Developer A’s work area, Developer A’s auxiliary sources are now linked to the wrong object. From Developer A’s perspective, it would have been better for Developer B to remove his old copy of the checked-in file in order to create a “dangling link” in Developer A’s work area. This might at least cause an overt error rather than an insidious source of data dictionary inconsistency.

This last illustration is contrived, of course, and any of several reasonable work area management conventions or policies would have sufficed to avoid such problems. For example:

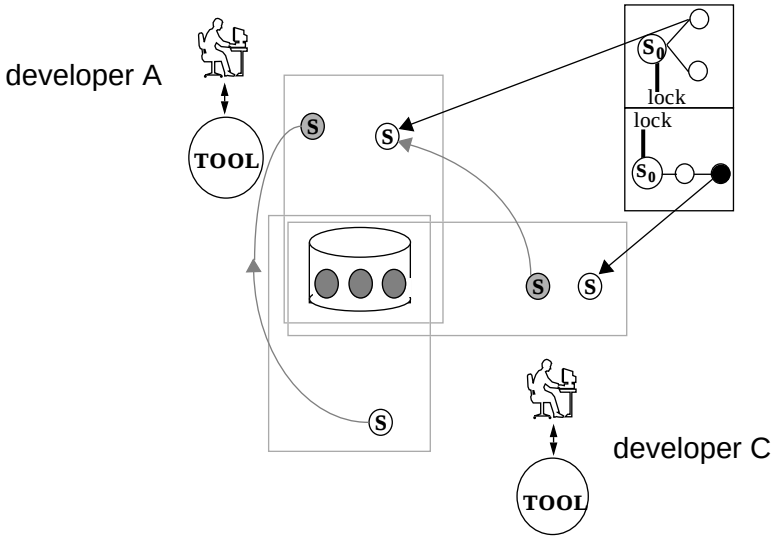

Figure 3-4c Shared Data Dictionaries

- Check-in operations result in a broadcast message to users of the checked-in object (as an auxiliary, or linked, object). The links can be re-established to refer to the repository versions.
- Check-out operations result in a broadcast message to potential users of the checked-out object so that new links can be established.
- “Link-chains” can be established by linking the last checked-in object to the newly checked-out object. Thus, in Figure 3-4c, Developer B’s old object would be deleted and replaced by a link to Developer C’s newly checked-out object.

Each of these alternatives has benefits and pitfalls. However, the point behind this extended and somewhat low-level discussion is that greater coordination among users—which is needed to reduce the cost of managing large derived objects such as data dictionaries—requires a degree of care in work area management in order to avoid excessive reprocessing of the data dictionary, excessive secondary storage costs, and costly errors caused by inconsistent data dictionaries or inconsistent views of data dictionaries.

3.2.4 Multiple repositories

The problems discussed in the previous scenario resulted from the intricate machinery introduced to maintain consistency among multiple copies of source across private work areas. An alternative approach is to have developers share the same work areas. This requires some form of synchronization, or locking, of source objects in the shared work area. One way to do this is to associate a local repository with each work area. In the following discussions, the term *workspace* will be used to mean a work area with an associated repository.

If the work area's repository is used to manage complete configurations (i.e., is used for more than locking), a collection of workspaces can be combined into hierarchies of configurations. Each work area can then be thought of as a scope in which changes can be made. In combination with scope, the use of links versus copying objects to "nested" workspaces can provide different degrees of visibility, stability and sharing among workspaces. Such a scenario is illustrated in Figure 3-5, which shows two child workspaces supporting different types of workspace insulation. Further, in this scenario each workspace can have associated with it different repository policies. For example, workspaces higher in a hierarchy may have stricter access privileges which reflect the effect of the wider scope of changes made near the top of a hierarchy.

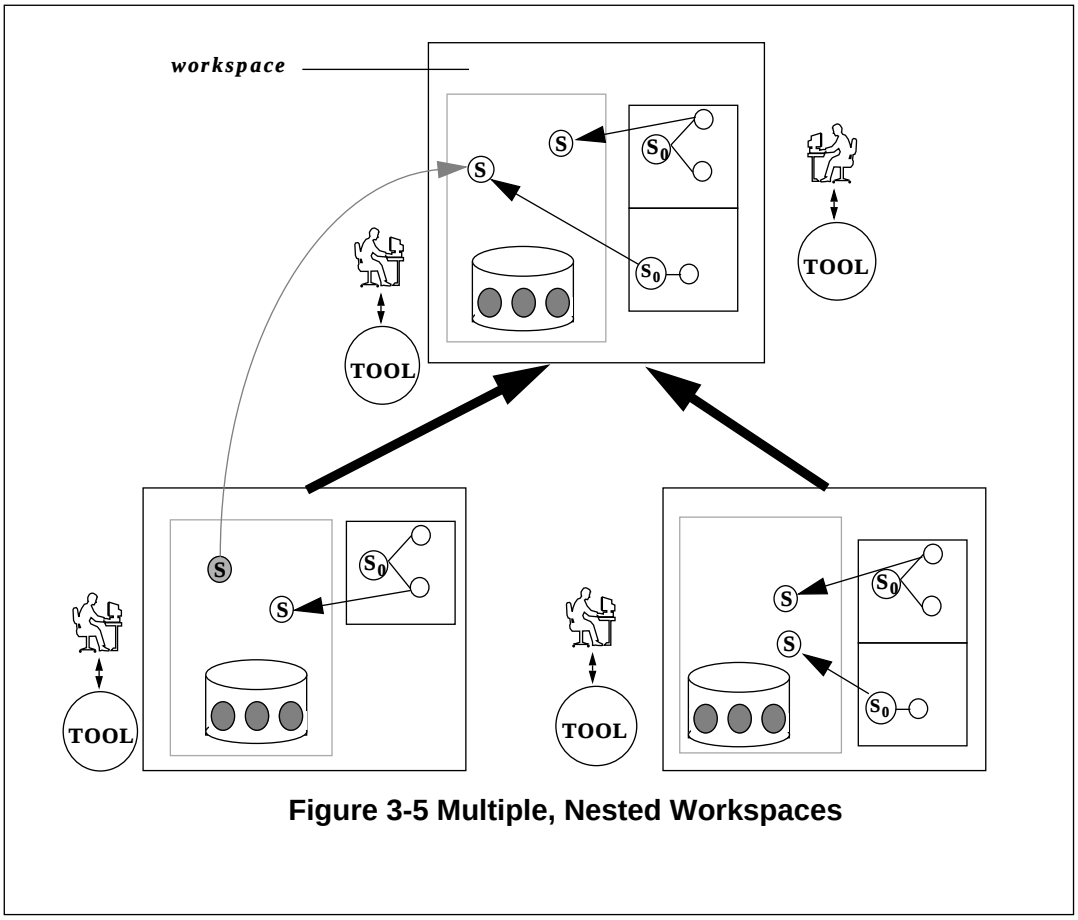

Figure 3-5 Multiple, Nested Workspaces

It is interesting to consider the fact that many of the advanced CM features provided by NSE [33] can be simulated readily using the techniques just described. Naturally, as the previous sections have illustrated there will be issues raised by any CASE/CM integration scheme, especially those that imply data duplication and/or require manual intervention.

3.2.5 Single repository, multiple partitions

One problem inherent in the multiple-repository approach just outlined is the non-locality of information on a configuration, or sets of versions of configurations, that defines the current status of a project. This is especially true if the creation of workspaces is under the control of developers, and not the configuration manager.

Instead of physically associating a unique repository with each work area, another approach is to partition a single repository into versions of configurations, with each work area being associated with one of these partitions. This relationship between a central CM repository and developer work areas is a key component of CM-managed developer workspaces. However, implementation constraints imposed by widely available version management systems such as RCS and SCCS may make such partitioning impractical without a concerted implementation effort to extend the primitive VM capabilities. CM systems such as SoftTool's CCC [11] and integration frameworks such as PCTE [5] do provide the foundations upon which managed workspaces can be more directly constructed.

3.3 Summary of CASE/CM illustrations

The contrived illustrations discussed in this section did not take into consideration real-world software process constraints, or implementation peculiarities of the host system, CM system or tool; and yet a number of intricate design implications were observed when combining even these simple tools and simple CM services.

The main points to note are that:

- There is no one "right" way to integrate CASE with CM. There are different cost/benefits associated with various approaches.
- Effective integration of CASE with CM requires that attention be given to issues of developer work area management.
- The benefits of developer isolation (e.g., work area stability) needs to be balanced against the costs of maintaining separate expensive derived objects, such as Ada program libraries.
- Sophisticated CM services can be modeled on, or constructed from, primitive services such as check-out/check-in. The use of conventions and manual processes can be effective in this respect.

This last point notes that widely available version management support software such as RCS [35] provides the foundations upon which sophisticated developer CM support can be constructed. However, sophisticated CM systems which directly support developer activities, or which can be tailored to support development activities, have already emerged in the marketplace. The following section discusses some of the issues that arise when considering the integration of CASE tools with such CM systems.

4 Integration of CASE with Advanced CM Systems

It is interesting to note that where version management services have in the past been specified for the purposes of integrating tools with CM, the services tend to be rather primitive, e.g., check-out/check-in, lock, etc. Examples of this tendency can be found in the PACT VM primitives [36], the CIS specification [9] and Hewlett-Packard's SoftBench message class for CM messages [7]. It is premature to conclude that consensus and commonality across tools and CM systems will be achieved through primitive services, and that integration is "easier" to achieve with primitive CM services than with more abstract CM services; however, this possibility needs to be considered when selecting sophisticated CM systems as a target for tool integration.

Feiler's CM paradigms [15] provide a context for discussing advanced CM systems. Feiler identifies four models of CM—long transaction, composition, change set and check-out/check-in. Check-out/check-in CM is the foundation case, and was discussed in the previous section. In the following brief discussion, some issues related to integrating CASE with the remaining three paradigms are outlined.

4.1 CASE and the long transaction model

The idea of a long duration transaction is not new, and has been explored both in CAD databases [20] and more recently in software engineering environments [3]. Appendix A has a detailed discussion of an experimental integration of a CASE tool with a commercially-available transaction-based CM system, NSE. The most significant issues which arose in that experiment that are likely to be related to transaction-oriented CM and not just to the NSE product include:

- The problem of inter-tool communication across transaction boundaries.
- The cost of duplicating large tool data repositories across multiple transaction instances.
- Establishing meaningful policies about which tools should be associated with which transactions.

A final point to note is that transaction-oriented CM systems are likely to rely upon operating system-level services to implement a *transparent repository*, i.e., each transaction operates within the same repository, but changes are local to the transaction until *committed* (at which point the changes are propagated to the enclosing transaction). The issue that may arise concerns the way in which tool data is made known to the CM transaction manager. In the NSE experiment, an implementation quirk of NSE allowed tool data to be associated with (and localized to) a transaction; however, the overhead of mapping tool data to an internal NSE data model of the configurations managed by the NSE transaction manager made it impractical to support full tool integration with the transaction mechanism. Similar restrictions may well arise in other transaction-based systems.

4.2 CASE and the composition model

The key features of the composition model of CM are the *system model* and the *selection rules*. The system model describes the relationships among versioned components, and the selection rules act as *predicates* to select valid—complete and consistent—subsets of the system model, i.e., configurations.

There are at least two issues which arise when integrating tools with a composition-based CM system:

- Maintaining consistency between (internal) tool system models and the CM system model.
- Naming of objects across tool and CM repositories to support uniform selection.

The first point reflects issues raised by the increased expressiveness, and semantic complexity, of the CM system model (as compared with simple versioning tools). As data is exported from a tool to the CM system, both the CM system model and its selection rules need to be updated to reflect changes made within the tool. The inverse is also true: the tool must be able to import data (or make use of derived data) that has been modified by another tool with resultant changes to the system model. For example, the SMARTSystem tool discussed in Appendix A has an internal system model based upon fine-grained objects (possibly at the level of program expression); its system model is clearly quite different from a CM system model which is based upon file-sized configuration items. Some kind of mapping between these system models is necessary. Fortunately, SMARTSystem also keeps track of changes at the file level, so this mapping is simple (although such simplicity can not be counted upon in general).

The second point reflects the general difficulties raised by the proliferation of private tool repositories in a CASE environment. Where tool data is simply exported to an external CM system, the introduction of private tool repositories simply adds additional complexity to the mapping function from tool to CM repository (i.e., in addition to version and system model mapping, some mapping between data models may also be necessary). However, in an environment where control integration predominates data integration (e.g., SoftBench [7], FIELD [31]), the naming of objects across tool repositories necessary to perform system selection can be a problem.

4.3 CASE and the change set model

The change set model is based upon the algebraic manipulation of sets of differences (“deltas”) that arise as a result of changes made to configuration items. For example, new systems can be “configured” through the application of specified sets of changes (hence the “change

set") to some known base system. The key questions that arise in integrating tools with a change set-based CM system are:

- What form will tool change sets take?
- How are change sets applied to tool data?

These two questions refer to the possibility of having an external agent (perhaps the CM tool) knowledgeable enough about the data management implementation of a tool to perform *differencing* operations between old and new versions of tool data. If an external agent can perform the differencing operation, then it may be possible for change set manipulation to occur without the active participation of the tool being integrated.

However, in general this will not be possible, especially where tools implement their own complex data management schemes. Instead, it may be necessary for the tool to export data which represents a difference between versions of the tool database. In this case, the form of the difference is opaque to the CM system. The generation of a new version of the tool data will require that the CM system make use of a tool-provided operation for change set manipulation, passing the (opaque) change-set as a parameter to this operation.

4.4 Summary of CASE and advanced CM systems

The classes of advanced CM systems discussed in this section provide greater degrees of support for developer CM (DCM). However, the greater sophistication of these CM systems may introduce added complexity in the integration of CASE tools with CM services.

5 Summary

CASE integration with CM can be thought of as a special case of tool-to-tool or tool-to-framework integration. A three level model of integration—processes, services and mechanisms—provides a context for identifying the factors that will contribute to a successful integration strategy:

- The process level introduces many factors which need to be addressed. Besides those process factors that are peculiar to CM, e.g., change control policies, other, more generic process issues must also be considered, including: integration with life-cycle models and steps within the life-cycle model, process-modeling notation and process evolution. Finally, it needs to be noted that most tools imply their own process constraints which must be considered when integrating tools with software processes.
- The service level introduces a logical layer between process and mechanism, and is a means for tool integrators to focus on *what* needs to be integrated instead of *how* to do the integration. The work of Feiler [15] and Dart [13] is particularly useful in identifying CM services that need to be integrated among tools and dedicated CM systems. Services for other engineering domains (e.g., design, test) are not as well established.
- The mechanism level characterizes the factors that affect the implementation of an integration solution. For CASE/CM integration, mechanism characteristics of tool architectures, CM architectures, and tool/CM integration architectures were discussed.

The relationships between processes, services and mechanisms can be complex. One consequence of this complexity is that there is no “right” way to integrate CASE with CM. Instead, CASE/CM integration must be viewed as a design problem with various trade-off decisions which must be made.

A key result of our study of CASE/CM integration is that the focus of attention should be on the relationship between the logical services provided by tools, and the software processes these services are intended to support. As a consequence, our emphasis has shifted away from the mechanisms of integration. This change in emphasis is particularly important for third-party tool integrators (including application developers who merely want to make use of CASE technology) who do not have access to internal tool mechanisms, and do not have the resources to commit to engage in a large, costly environment integration effort.

Related and Future Work

Two related efforts are currently underway in the SEI Software Development Environments Project. The first is to more fully formalize the services implicit in Dart’s spectrum of CM concepts [13]. The second is to expand experimentation with CASE/CM integration across a broader span of life-cycle processes than addressed in the integration experiment discussed in Appendix A of this report.

An effort closely related to the CASE/CM integration work is the Next Generation Computing Resources (NGCR) Project Support Environment Standards Working Group (PSESWG). In the NGCR/PSESWG, a reference model for comparing software development environments and identifying interface areas ripe for standardization is being developed. This reference model ([29], [30]) draws heavily upon the three-level model of integration and the concept of *service profiling* discussed in this report.

Acknowledgments

Many thanks to Fred Long for his detailed and accurate comments on this report, and to Peter Feiler and Alan Brown for their time and assistance in helping to develop and clarify the ideas described in this report. Finally, my personal thanks to all the members of the Software Development Environments Project and the CASE Technology Project for making my stay at the SEI so enjoyable and productive.

References

[1] *A Reference Model for Frameworks of Software Engineering Environments (Version 2)* ECMA & NIST, 1991. ECMA Report Number TR/55 Version 2, NIST Report Number SP 500-201.

[2] *A Reference Model for Project Support Environment Standards.* U.S. Navy NGCR Program, 1992. Draft Technical Report.

[3] Barghouti, Nasar, Kaiser, G. *Concurrency Control in Advanced Database Applications.* Technical Report CUCS-425-89, Columbia University, Dept. of Computer Science, May 1990.

[4] Boehm, B. "A Spiral Model of Software Development and Enhancement." *IEEE Computer.* 61-72, 1981.

[5] Boudier, G., Gallo, T., Minot, R., Thomas, I. "An Overview of PCTE and PCTE+." *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Engineering Environments*. Boston, MA, 1988.

[6] Cagan, M.R. *Forms of Software Tool Integration.* Technical Report Tool Integration Report IDE-TI-90-04, Interactive Development Environments, 595 Market Street, 10th Floor, San Francisco, CA 94105.

[7] Cagan, M.R. "The H.P. SoftBench Environment: An Architecture for a New Generation of Software Tools." *Hewlett-Packard Journal, 4*1(3), June 1990.

[8] *CASE Communiqué Charter and Organization* (Lisa Stroyan, Editor), Hewlett-Packard, Boulder, CO, 1991.

[9] *CASE Interface Services Base Document,* Digital Equipment Corporation, Nashua, NH, 1990.

[10] Brown, A., Feiler, P., "Wallnau, K. Past and Future Models of CASE Integration." To appear in the *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*. Montreal, Canada. July 1992.

[11] *CCC: Change and Configuration Control Environment* SoftTool Corporation, 340 South Kellogg Ave., Goleta, CA 93117, 1987.

[12] Ploedereder, E., Fergany, A. "The Data Model of the Configuration Management Assistant." *Proceedings of the 2nd International Workshop on Software Configuration Management*. ACM SIGSOFT, October 1989.

[13] Dart, S. "Concepts in Configuration Management Systems." *Proceedings of the Third International Conference on Software Configuration Management.* ACM, Trondheim, Norway, June 1991.

[14] *DOD-STD-1838A, Common Ada Programming Support Environment (APSE) Interface Set (CAIS), Revision A,* U.S. Department of Defense, 1988.

[15] Feiler, P. *Configuration Management Models in Commercial Environments.* Technical Report CMU/SEI-91-TR-7, DTIC: ADA235782, Software Engineering Institute, Carnegie Mellon University, March 1991.

[16] Feiler, P., Downey, G. *Transaction-Oriented Configuration Management: A Case Study.* Technical Report SEI-90-TR-25, DTIC: ADA235639, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[17] Fleming, R., Wybolt, N. *Frameworks for CASE Tool Integration.* Technical Report, CADRE Technologies, Inc., Providence, RI, 1990.

[18] *IEEE Standard for Software Configuration Management Plans,* The Institute of Electrical and Electronics Engineers, Inc. (IEEE), 345 East 47th Street, New York, 10017, 1983. ANSI/IEEE Std. 124-1983.

[19] Katz, R. “Toward a Unified Framework for Version Modeling in Engineering Databases.” *ACM Computing Surveys*. 1990.

[20] Kim, W., et. al. “A Transaction Mechanism for Engineering Databases.” *Proceedings of the 10th International Conference on Very Large Databases*. August 1984.

[21] Korn, D.G., Krell, E. “A New Dimension for the UNIX File System.” *Software Practice and Experience,* Vol. 20, June 1990.

[22] Leblang, D., Chase, R. “Computer-Aided Software Engineering in a Distributed Workstation Environment.” *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Engineering Environments*. Pittsburgh, PA, April 1984.

[23] Mercurio, V.J., Meyers, B.F., Nisbet, A.M., Radin, G. “AD/Cycle Strategy and Architecture.” *IBM Systems Journal,* 29(2), 1990.

[24] Morris, E., Feiler, P., Smith, D. *Case Studies in Environment Integration.* Technical Report CMU/SEI-91-TR-13, Software Engineering Institute, Carnegie Mellon University, July 1991.

[25] Nejmeh, B. *Characteristics of Integrable Software Tools.* Technical Report INTEG_S/W_TOOLS-89036-N Version 1.0, Software Productivity Consortium, May 1989.

[26] Paulk, M.C., Curtis, B., Chrissis, M.B., et al. *Capability Maturity Model for Software.* Technical Report CMU/SEI-91-TR-24, DTIC: ADA240603, Software Engineering Institute, Carnegie Mellon University, August 1991.

[27] *Portable Common Tool Environment (PCTE) Abstract Specification*, European Computer Manufacturers Association (ECMA), 1990, ECMA-149.

[28] *Proceedings of the 5th International Software Process Workshop.* IEEE Computer Society Press, 1989.

[29] Brown, A. and Feiler, P. *A Project Support Environment Services Reference Model.* Technical Report CMU/SEI-92-TR-2, Software Engineering Institute, Carnegie Mellon University, 1992.

[30] Brown, A. and Feiler, P. *An Analysis Technique for Examining Integration in a Project Support Environment.* Technical Report CMU/SEI-92-TR-3, Software Engineering Institute, Carnegie Mellon University, 1992.

[31] Reiss, S. "Interacting with the FIELD Environment." *Software Practice and Experience,* Vol. 20, June 1990.

[32] Strelich, T. "The Software Life Cycle Support Environment (SLSCE): A Computer-Based Framework for Developing Software Systems." *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Engineering Environments*. Boston, MA, 1988.

[33] *The Network Software Environment.* Sun Technical Report, Sun Microsystems, Inc., 1989.

[34] Thomas, I., Nejmeh, B. *Tool Integration in a Software Engineering Environment.* Technical Report SESD-91-11, Revision 1.1, Hewlett-Packard.

[35] Tichy, W.F. *Design, Implementation and Evaluation of a Revision Control System.* Technical Report CSD-TR-397, Dept. of Computer Science, Purdue University, 1982.

[36] *Version Management Common Services* G.I.E. Emeraude, 38 Bd Henri Selier, 92154 Suresnes, France, 1988.

[37] Brown, A.W., Feiler, P.H., Wallnau, K.C. "Understanding Integration in a Software Development Environment." (To appear in) *Second IEEE International Conference on Systems Integration*. Morristown, NJ, June 1992.

[38] Wasserman, A. "Tool Integration in Software Engineering Environments." Fred Long (Editor), *Lecture Notes in Computer Science*, Vol. 467, Springer-Verlag, 1990.

Appendix A Extended Illustration of CASE/CM

A.1 Introduction

The experimental integration of SMARTSystem with alternative CM systems provided a basis for identifying and exploring issues of CASE integration with CM. This appendix provides some of the grim details of the experimental integration of ProCASE's SMARTSystem with Sun Microsystem's Network Software Environment (NSE). The purpose of exposing these details at this level is to illustrate the sometimes subtle interplay among processes, services and mechanisms in even relatively straightforward integration efforts.

SMARTSystem provided a useful example of a CASE tool for the purposes of this experiment because it has a number of interesting characteristics:

- It is a member of the database tool class; this presents a complex set of integration issues.
- It provides CM primitives for multi-user and team support; this presents interesting CM-to-CM integration problems.
- It has a relatively "open" architecture; the SMARTScreen[12] interpreter provides a limited form of access to tool source code, i.e., many behavioral aspects of the system are under the control of the user. However, the tool is still "egocentric" in that it will not respond to stimuli originating from other tools.

NSE provided a useful CM system for this experiment since it, too, has a number of interesting characteristics:

- Two NSE features—the transparent repository and separation of CM data model (i.e., the component hierarchy) from data management (i.e., TFS)[13]—support "easy" integration of UNIX-based tools with NSE.
- NSE supports a relatively pure transaction model [33]; therefore, integration of tools with NSE may reveal interesting interactions between CASE tool and transaction-based team coordination.
- NSE provides significant functionality for developer CM, with decreasing support for project CM, corporate CM and application CM. This is the same pattern that appears in many CASE tools, and a different pattern than provided by other CM systems.

Structure of this Appendix

Section A.2 describes the experiment scenario. Section A.3 describes the integration strategy in detail. Section A.4 discusses some architectural issues which arose as a result of the integration strategy adopted. Finally, Section A.5 summarizes the results of the experiment.

12. SMARTScreen is an interpreted user-interface language which also provides object management system access services.

13. The NSE component model and TFS will be described later in this appendix.

A.2 Scenario

This section describes the project scenario for the experiment. Section A.2.1 describes the product being managed by SMARTSystem and NSE, and Section A.2.2 describes a skeletal software process.

A.2.1 Product description

For the purposes of the experiment, a small but non-trivial application was managed by NSE and SMARTSystem. The GNU *RCS* system was chosen because it has a moderately-sized product structure, and because it is a less-than perfect match for NSE.

GNU *RCS* is composed of two systems: the GNU *RCS* system, and the GNU *diff* system (used by GNU *RCS*). GNU *RCS* consists of 20 source (".c") and 2 include (".h") files totalling approximately 13K lines of source code, and 9 executable programs (*ci, co, ident, merge, rcs, rcsdiff, rcsmerge, rlog, and rcstest*). GNU *diff* consists of 15 source and 4 include files totalling approximately 10K lines of source code, and two executable programs (*diff* and *diff3*).

The GNU *RCS* system can be configured to use the traditional UNIX versions of *diff* and *diff3*, or it can be configured to use the GNU versions of *diff* and *diff3*. The GNU *diff* programs offer enhanced functionality which results in greater *RCS* functionality. For the purposes of the integration experiment, both the traditional- and GNU-*diff* configurations will be managed. This resulted in different configurations which are selected at compile time.

A.2.2 Process description

Since the GNU *RCS* and GNU *diff* products are of moderate size, a process scenario with two developers being responsible for maintaining the system was reasonable. In this scenario two developers, Mark and Sally, maintain GNU *RCS* and GNU *diff*. For purposes of context, assume there is a general GNU installation of which *RCS* and *diff* are just two components; also, assume that Sally is the GNU manager, and Mark is a junior programmer.

For simplicity, the process will be decomposed into these activities: change initiation, change execution, change testing, integration testing, and release.

A.2.2.1 Change initiation

The GNU manager, Sally, is responsible for analysis of feature and bug requests, and determining what changes need to be made to the GNU software. Sally assigns change tasks to herself and to Mark.

In the experiment, changes were:

- Non-overlapping and non-interacting.
- Non-overlapping but will interact.
- Overlapping, and requiring some kind of merging or synchronization.

The change request is assigned to either or both Mark and Sally. Each change request has a designated *owner*. Sally is responsible for *promotion* of a change to the next step in the process regardless of who “owns” the change request.

A.2.2.2 Change implementation

Despite the moderate size of the product, each software enhancement is made in an isolated context. Since change requests can be assigned to either or both Mark and Sally, coordination must take place in some cases as developers share the same source. Change implementation is completed when the changed component successfully “builds.”

A.2.2.3 Change testing

For each change request, the designated task owner must construct and execute a test program in isolation for verifying the change. This test must be added to the *RCS test suite*, which must execute satisfactorily as a whole. The test suite is managed as part of the *RCS* configuration. Change requests which result from bug reports must result in the addition of the bug (if reproducible) to a *regression test suite*. The regression test suite is also a managed configuration item. The regression suite also must execute satisfactorily as a whole.

A.2.2.4 Integration testing

Changes that have successfully passed change testing are considered ready for integration with other changes that may have been made. Sally is responsible for integration testing of all enhancements. Integration is achieved when the *RCS* test suite (including tests for new features) and regression tests have successfully completed.

A failure at integration testing may result in the “demotion” of a change, or may be resolved by in-place repairs.

A.2.2.5 Release

Release occurs after successful integration testing. Since GNU is publicly available software, the release process is simply the installation of the GNU software to a publicly accessible location. Notification of potential users of GNU *RCS* and GNU *diff* will be made upon successful installation.

A.3 Integration strategy

A number of issues contributed to the strategy for integrating SMARTSystem with NSE. The issues which appeared to be paradigmatic to tool integration with CM appear in the main body of this report. This section (A.3) describes the most significant design issues raised in the experiment. Section A.3.1 discusses the effect of CM application areas on the experiment (refer to Section 2.1.1). Section A.3.2 discusses the delegation of services between SMARTSystem and NSE (refer to Section 2.4). Finally, Section A.3.3 describes in detail the integration of SMARTSystem and NSE-specific services and mechanisms.

A.3.1 CM application focus

For the purposes of this experiment, the main focus was placed on developer CM. Issues related to project CM life-cycle support and process-related issues of corporate CM were not directly addressed. The focus on developer CM resulted in the emphasis on CM support for individuals and small teams.

The decision to emphasize developer CM was justified because both NSE and SMARTSystem provide substantial developer support; similarly, both systems provide ever-decreasing support for project, and corporate CM (see Figure 2-5). Support for these CM process views could be constructed from SMARTSystem and NSE services, but this would have shed more light on process-encoding in SMARTSystem and NSE than on CASE/CM integration. The small-team process scenario induced by the GNU product also justified a focus on developer CM.

Thus, a combination of factors such as product qualities (of GNU *RCS*) and NSE and SMARTSystem resulted in a design decision to emphasize developer CM.

A.3.2 Tool/CM services profiling

Since SMARTSystem and NSE both support developer CM there was considerable overlap in the services offered by each system. The integration strategy addressed this overlap by identifying delegated and shared responsibilities. The services profiles illustrated in Figures 2-5 and 2-6 earlier in this report are illustrative of how service delegation was determined.

A.3.2.1 SMARTSystem individual support, NSE team support

After evaluating SMARTSystem and NSE services in support of the individual and team views of CM, it was determined that individual support services would be delegated to SMARTSystem while team-support services would be delegated to NSE. This decision led to a design where NSE was used as the primary repository, while SMARTSystem databases were treated as transient, derived objects.

Several factors contributed to this decision:

- SMARTSystem has a far richer set of primitives supporting individual C programmers than NSE. For example, although the NSE refinements of UNIX *make* provide enhanced build capabilities when compared with “vanilla” UNIX *make*, SMARTSystem’s abilities to model C source files as modules allows for significant optimizations to the build process.
- NSE has a far richer set of CM primitives than SMARTSystem for supporting families of systems, such as the ability to model variants. See Appendix B of this report for a synopsis of SMARTSystem CM primitives.
- The NSE repository is more generalized, and supports integration of a broader class of UNIX tools than does the SMARTSystem object management system (which is, in fact, opaque to external tools). These tools, such as debuggers, may be useful in maintaining the GNU source.

A.3.2.2 Design consequences of allocation of responsibilities

The decision to use NSE as the principle repository had two design consequences. One concerns the inconsistency that can arise between SMARTSystem and NSE repositories. Another concerns the structuring of the NSE repository to accommodate SMARTSystem. These consequences are discussed below.

Repository consistency consequences

Since transient SMARTSystem repositories contained copies of configurations that also exist within the persistent NSE repository, maintaining consistency among these repositories became an essential design objective.

Repositories can become inconsistent when:

- Changes are made within a SMARTSystem workspace and are then "checked-in" to the SMARTSystem database. At this point, the SMARTSystem database is a revision of the configuration that exists in NSE.
- Changes are made to source in an NSE environment[14] from which a SMARTSystem database has been derived. At this point, NSE is managing a revision of a configuration that exists within SMARTSystem.

There are several kinds of repository data which may become inconsistent:

- source code (".c", ".h" files)
- test scripts
- system model (derived object dependencies, derivation options/flags, system component objects)
- derived objects (".o" files and executables)

Repository consistency was a major theme in the integration of workspace and transaction services supported by both SMARTSystem and NSE.

NSE environment structure

Figure A-1 illustrates the environment structure of the GNU products in NSE. There were a variety of ways to structure a family of NSE transactions to support the experiment scenario. The one illustrated in Figure A-1 and described below represents a compromise between the life-cycle level models described by Sun in [33] and models optimized for use with SMARTSystem.

The top-level environment is GNU. GNU corresponds to a UNIX directory which becomes the NSE *control point*[15] for GNU environments. Within the GNU environment there are several *components,*[16] each corresponding to a GNU system, e.g., *rcs* and *diff*. The top-level environ-

---

14. The term NSE *environment* and NSE *transaction* in NSE are used synonymously in this report.

15. A *control point* defines the part of the UNIX file system that is controlled by an NSE transaction.

16. A *component* in NSE is an aggregate object. NSEI transactions manage changes to components.

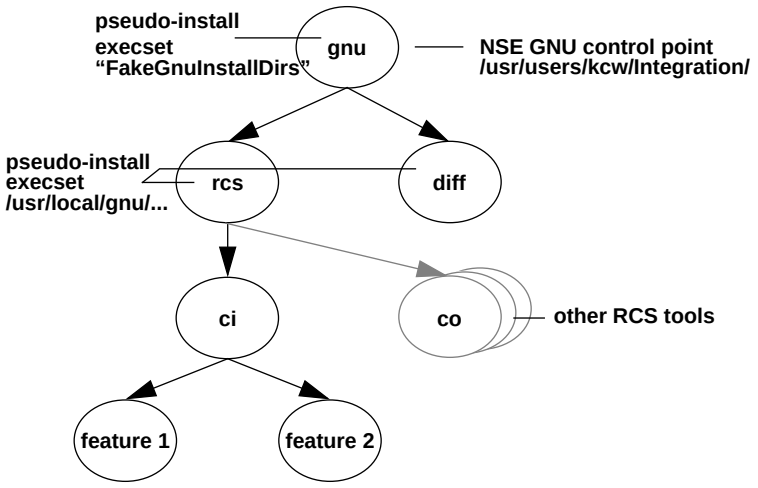

Figure A-1 NSE Environment Structure

ment, named “gnu,” represents the GNU repository; as such it is a non-terminating transaction. The gnu environment has components for each GNU system, and a *pseudo-target*[17] for the installation of each system.

Child environments of gnu are created by *acquiring*[18] the associated top-level component. That is, the *rcs* environment contains the acquired “:gnu:rcs” component. These environments represent the system integration environments. The life-span of these environments is on a per-release basis. That is, as new GNU releases become available, new integration environments are established from the gnu environment and the GNU release is built in the integration environment. Successful *builds* result in a *reconcile*[19] back to the GNU repository. Enhancements, patches, etc., to the latest GNU release are done in environments descended from the integration environment.

The GNU *RCS* and GNU *diff* systems are composed of multiple, related executable programs. For these experiments, each of these executables, which are modeled as *targets*[20] in the NSE component model, became the sole members of like-named components. Thus, the *RCS ci* program became the target of the component named “:gnu:rcs:ci.” This model, depicted in Figure A-2, allowed developers to *acquire* a minimal configuration needed to construct individual programs in a GNU system. This makes it easier to construct minimal SMARTSystem data-

17. A *pseudo-target* is an imaginary object used to control software manufacture, or software build.

18. The *acquire* operation makes components available to child, or nested, transactions.

19. The *reconcile* command makes changes made to an NSE component visible to the parent transaction (i.e., it acts like a “commit” without terminating the child transaction).

20. A *target* in NSE is a kind of *component* which is constructed through a build process. Pseudo-targets can be modeled as targets.

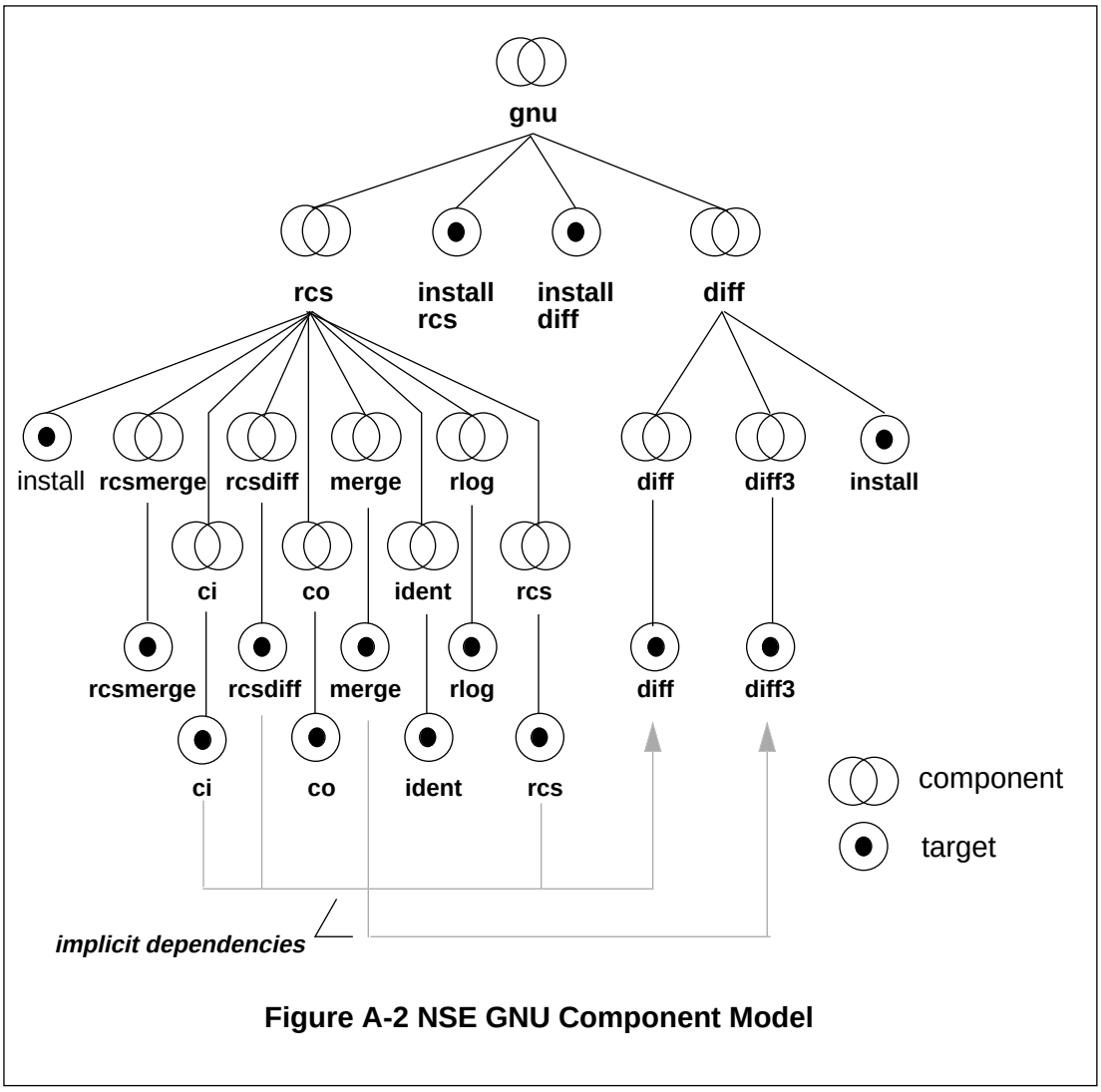

Figure A-2 NSE GNU Component Model

bases, thereby mitigating the large disk space consumption of SMARTSystem databases. Unfortunately, this further decomposition adversely affects unit testing within the lowest-level NSE environments. That is, a program like *ci* cannot be tested without a surrounding context of other *RCS* commands such as *co* and *rcsdiff* from the same system release. Since only the source which is a part of the *target* will be *acquired* into the environment, some means other than the NSE *acquire* must be used to establish a testing context.

NSE *translucent execsets*[21] can be used for testing purposes. Assume the GNU systems are to be installed in the UNIX directories "/usr/local/gnu/bin", "/usr/local/gnu/lib", and "/usr/local/gnu/man." These directories can be added to the GNU system integration environments as *execset* directories. The integration environments can then pseudo-install the experimental release into the execset directories. Installing into translucent execsets allows all other direc-

21. *Execsets* in NSE allow different transactions to have different read-only file systems mounted. The intent of execsets was to support automatic selection of appropriate tools, e.g., Sun3 vs. Sun4 C compilers, based upon attributes of the transaction. A discussion of *translucent* execsets is thankfully beyond the scope of this report.

tory contents within the execset directories to remain visible. Child environments of the integration environment will inherit the execset, and testing can take place as if it were occurring in an installed setting. Thus, simple manipulation of the UNIX path variable will direct the test harnesses to use binaries local to the unit test environment before using the pseudo-installed binaries found in the inherited execsets.

The above discussion on *execsets*, testing and storage optimization illustrate the complex interplay among software mechanisms, software process requirements and integration pragmatics on the design of in integration solution.

A.3.3 Services integration

The most important services provided by both SMARTSystem and NSE are the transaction and workspace services. The following sections describe in detail how these services were integrated in this experiment.

A.3.3.1 Transactions in SMARTSystem and NSE

An excellent description of the details of NSE transactions can be found in [33] and [16]; these details will not be repeated in this report.

SMARTSystem also supports a transaction model, though this model is significantly different, and more restrictive, than the NSE model. SMARTSystem transactions are based upon several concepts and primitives:

- Fine-grained locking enforces pessimistic transactions, but the fine-granularity mitigates the cost of pessimism.
- The SMARTSystem database evolves as a whole, i.e., the database represents one configuration (see Appendix B).
- SMARTSystem workspaces coordinate with a shared database, not directly with each other; thus SMARTSystem transactions are not nested, but strictly one-level.
- SMARTSystem transactions result in a linear development path, i.e., no variants of individual objects or of configurations.

Figure A-3 illustrates the interaction of SMARTSystem workspaces, fine-grained locking and the SMARTSystem database; these interactions constitute transactions in SMARTSystem.

Figure A-3 depicts the life-cycle view of a SMARTSystem database, i.e., the evolution of a database through the lifetime of a project. In this model, a SMARTSystem database evolves as a result of two types of transactions: internal and external transactions. Internal transactions affect the state of the SMARTSystem database; external transactions affect the relationship of SMARTSystem databases to the enclosing software environment.

The salient points of Figure A-3 are:

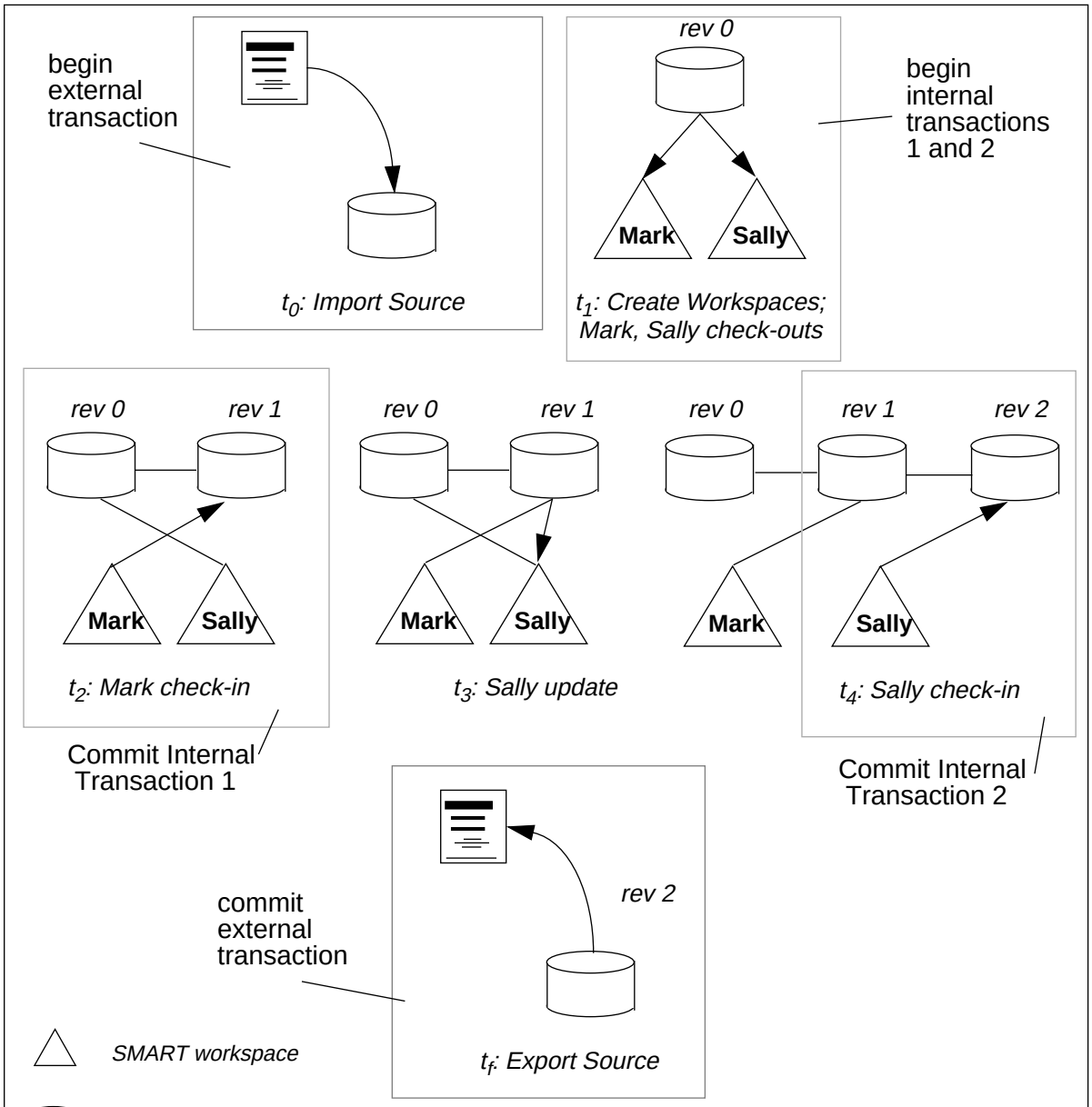

Figure A-3 SMARTSystem Transactions

- Fine-grained locking ensures that the check-outs occurring at time t_1 are non-overlapping; therefore the workspace update operation at time t_3 is non-destructive, and thus no merging is necessary.
- Each internal transaction results in a new revision of the shared database; the result is a linear development path.

- The duration of internal transactions is defined by the time between the last check-out of an object into a workspace, and the first check-in from that workspace. That is, while fine-grained objects can be checked-out, only whole configurations may be checked-in.
- The duration of external transactions is defined by alternating imports and exports. Note that while entire workspaces must be checked-in, databases may export a subset of managed sources.

A.3.3.2 Transaction integration

NSE transactions are defined by bracketed *acquire-reconcile* pairs. In this experiment, SMARTSystem transactions occurred within the context of single NSE transactions. A state-machine view of this integration strategy is depicted in Figure A-4. The arcs between states indicate actions that are taken during state transitions; state names indicate the relationship between transaction states and the software process defined in Section A.2.2 (i.e., state 2.2.1 corresponds to "change initiation," as described in Section A.2.2.1).

The salient points of this integration are:

- The states designated as ***I*** (for "inconsistent") indicate inconsistency between the NSE repository and SMARTSystem. Each of these states immediately reverts to its predecessor state.[22] Inconsistency caused by changes within NSE (flagged by NSE *change notification*[23] messages) reflects parallel development among different SMARTSystem databases. The easiest way to deal with these inconsistencies is to ignore them, and rely instead on NSE-supplied merge services. Inconsistency caused by changes to system-model related information within SMARTSystem is more difficult to deal with, at least in an automated way. Such changes could precipitate changes to the NSE *component model*[24] and Makefiles (e.g., where new files are added) or to hidden dependency files managed by NSE (e.g., command-line parse-flags used to build a system from within SMARTSystem). More work needs to be done to address this form of inconsistency.
- An arbitrary number of internal SMARTSystem transactions are allowed to occur within the context of a single NSE transaction. Therefore, only a single SMARTSystem database will be associated with any NSE environment. This represents a constraint on the integration of NSE and SMARTSystem workspaces (discussed in the following section).

A.3.3.3 Workspaces in SMARTSystem and NSE

Transactions and workspaces are closely related in both NSE and SMARTSystem. While no formal definition of workspace can be provided in this report (since none yet exists), the role workspaces play in developer CM support can be described, at least in the context of this experiment.

---

22. This part of the experiment was incomplete. In a real scenario, some action would need to be taken to re-establish consistency.

23. NSE supports a limited notion of notification based upon the occurrence of specific kinds of events (e.g., reconcile).

24. The NSE *component model* is a data model which maps a configuration model (consisting of NSE *targets, components*, etc.) to actual contents in the UNIX file system.

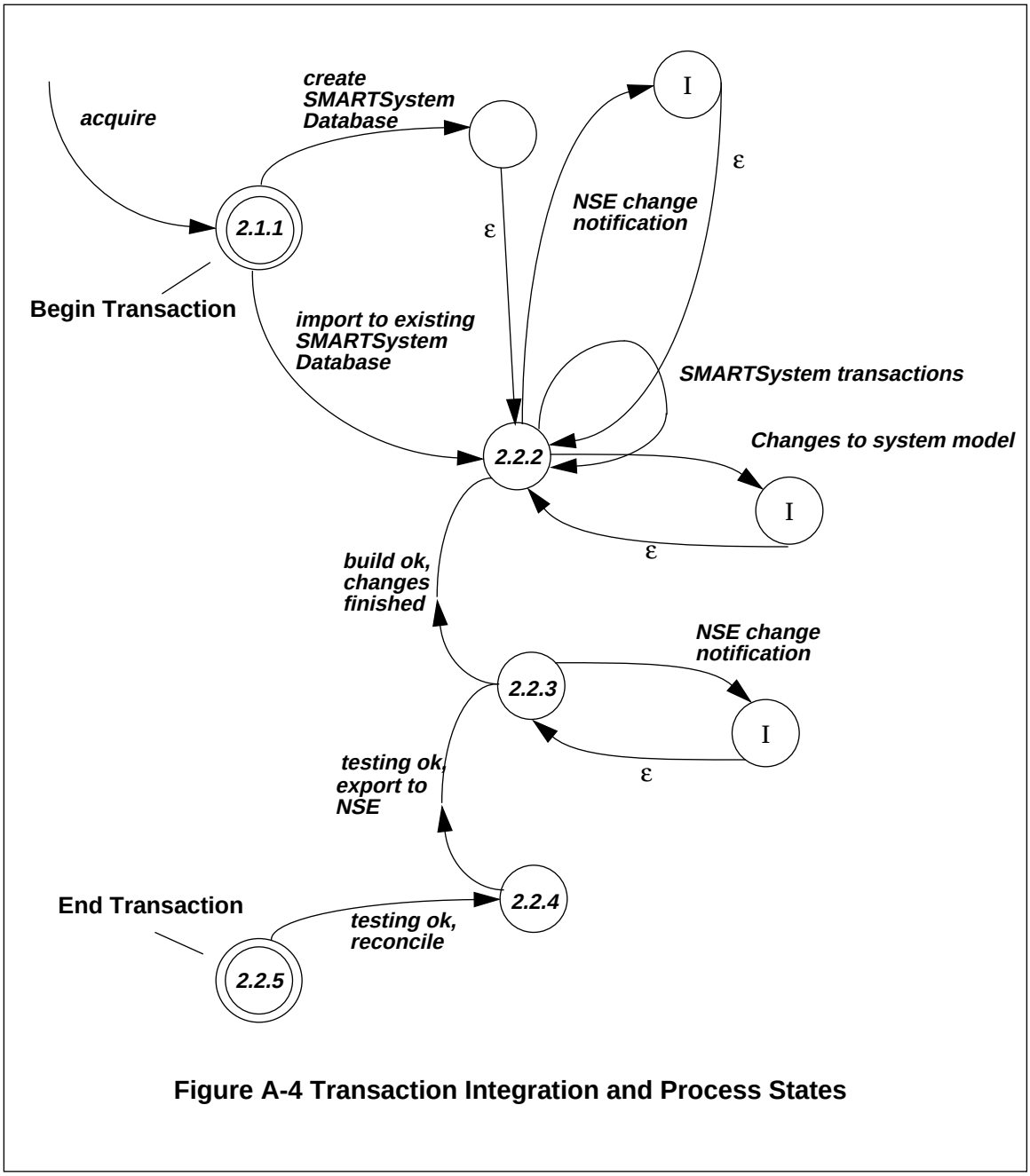

Figure A-4 Transaction Integration and Process States

Workspaces support individual developers and teams of developers by providing support for two opposing, but complementary concepts: *insulation* and *communication*.

Workspace insulation in NSE and SMARTSystem

Insulation ensures a stable context in which individuals can perform builds and tests. Without insulation, changes introduced by other developers would force continuous integration, and complicate the process of introducing and testing isolated functionality.

Two aspects of insulation are of interest: insulation from changes made outside of the workspace context (external insulation), and insulation from changes made within a workspace.

1. The SMARTSystem workspace supports external insulation by creating a copy of the database within the workspace, and by supporting a pessimistic transaction scheme. This transaction scheme is based on fine-grained locking, and guarantees that multiple copies of the database (in multiple workspaces) can be non-destructively merged. Since only one individual can be active within a workspace at any time, internal insulation in SMARTSystem is not an issue. That is, the SMARTSystem workspace has files that can be shared by users, but also has "swap" files created for each transient SMARTSystem client (see Section 2.3.1.2) that cannot be shared; hence workspaces remain private.

2. The NSE workspace supports external insulation through the transparent repository. Changes that appear to be made within the repository actually occur in a TFS[25]-based version of the repository. Since each environment has its own repository, insulation is guaranteed. Internal insulation in NSE is supported by version control primitives that lock file-sized objects. Since NSE workspaces are essentially UNIX directories,[26] UNIX permissions define the basic access control primitives.

Workspace communication

This refers to the way changes are coordinated between workspaces. There are two aspects of communication: notification and propagation.

1. **Notification**. Notification is the means of alerting users and tools of some important event, or change in status within a workspace. Notification can be used to coordinate users sharing a single workspace or to broadcast important events to multiple workspaces. In the latter case, notification is a process-oriented hedge against the risks of too much insulation.

   NSE supports notification by permitting NSE users and administrators to associate executable programs (e.g., shell scripts) with various important events that occur within NSE environments (e.g., reconcile events).

   SMARTSystem supports notification through its programmable user interface language, SMARTScreen. Any action taken by users at the user interface level can trigger notifications (i.e., event-driven triggers). SMARTSystem also provides triggering on changing property values (i.e., data-driven triggers).

---

25. TFS—the *Translucent File Service.*

26. Other characteristics of NSE workspaces are execsets, variants, locks, etc., but these do not pertain directly to workspace insulation.

2. **Propagation.** Propagation is the means by which changes are made available to other workspaces. There are three aspects of propagation: scope, visibility and migration.

    a. **Scope**. Workspace scope is analogous to programming language scope. Scope defines a context in which an object *may* become visible. Workspace scope differs from programming language scope in that different copies (versions, revisions, etc.) of an object may exist within a single scope.

    NSE's hierarchical environment structure provides multiple levels of scope; the scope of an object in NSE is an environment and its descendent environments.

    SMARTSystem provides only global scoping. Since SMARTSystem workspaces are private, only the database can define a scope.

    b. **Visibility**. By changing the visibility of configured items to other workspaces, e.g., through use of a "latest-version" token, changes can be made more widely available. Note the interaction of visibility with scope.

    Visibility in NSE is defined through the use of the NSE version control system (VCS) and *preserve*[27] within environments. NSE supports two forms of visibility, both of which can effect configuration consistency: latest members consistency, and shared members consistency. The distinctions between these forms are discussed in [33].

    Visibility in SMARTSystem is a degenerate concept, since there is only global scope.

    c. **Migration**. This is the more traditional method of propagating changes, and is exemplified by the "check-in/check-out" model. Change sets [15] offer a different form of migration.

    NSE supports change migration through the transaction mechanism. That is, transactions in NSE are a form of workspace communication. Interestingly, in NSE it is not possible to propagate changes through migration without also changing the scope of these changes. That is, to migrate a change to a sibling environment/transaction, the change must be propagated to the scope which is shared by the siblings. The "scoped workspace" model discussed later in this report addresses this issue.

    SMARTSystem supports change migration through the classical check-in model. However, the entire contents of a workspace are checked in to the global scope. See Appendix B for details.

---

27. *Preserve* is a NSE command which acts like a snapshot of a transaction—it saves changes within a transaction for later rollback, but does not propagate these changes to the enclosing transaction.

A.3.3.4 Workspace integration

There are two workspace integration schemes described below: the nested SMARTSystem workspace scheme, and the scoped SMARTSystem *database* scheme. Before discussing these schemes two points must be made.

First, in the following discussions the terms “database” and “workspace” will both be used when describing SMARTSystem’s workspace concept. In fact, the SMARTSystem database and workspace concepts are inextricable. All workspaces are rooted to a database server, which manages a set of database files; further, each workspace (i.e., database client) manages its own set of files. Thus, integration of SMARTSystem workspaces requires the integration of the database files as well as workspace files.

Second, the per-process nature of NSE’s transparent repository means that the SMARTSystem database server and clients must always execute within the same NSE environment in which they were created. This is illustrated in Figure A-5. Thus, it is critical that SMARTSystem

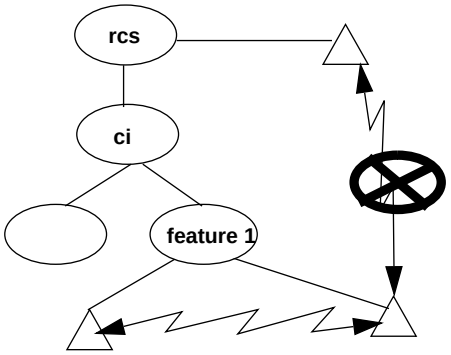

Figure A-5 Valid and Invalid Inter-Workspace Communication

and NSE workspaces integrate so that SMARTSystem database servers and clients always access database files that have been derived from the currently active NSE environment.

Nested SMARTSystem workspaces

Figure A-6 illustrates one way of associating SMARTSystem databases and workspaces with NSE environments. In this model, each NSE environment will have one or more SMARTSystem database associated with it. Each SMARTSystem database in turn supports one or more SMARTSystem workspaces, which are also associated with the NSE environment.

The way these associations between SMARTSystem and NSE environments are made is trivially simple: the SMARTSystem database and workspace files are created under the control point of the parent NSE environment while the environment is activated. These SMARTSystem files will be opaque to NSE transaction/CM services, but will be associated with the environment through the TFS.

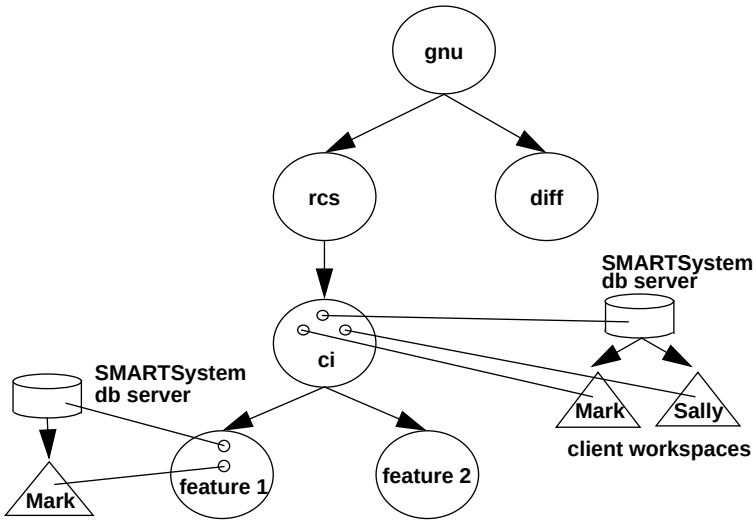

Figure A-6 SMARTSystem Sub-Workspaces

This approach takes advantage of the separation of the NSE data model and data management services. That is, by bypassing the component model and using TFS directly, NSE can manage the versioned repository while having other non-versioned objects go along for the ride on TFS remotely-mounted directories. Thus, SMARTSystem database and workspace files can become visible only in the activated NSE environment in which they were created.

There are some potential benefits to associating SMARTSystem databases with single NSE environments. Besides the simplicity in maintaining consistency between NSE and SMARTSystem repositories, SMARTSystem can coordinate multiple users within an NSE environment. Since no support beyond file-level VCS locking is supplied for parallel development in NSE, transactions are optimistic across NSE environments, but pessimistic within NSE environments. SMARTSystem also requires locking, but since it provides locking at a much finer granularity than Unix files, a greater degree of parallelism is theoretically attainable. This illustrates an advantageous integration of otherwise overlapping services.

There are costs associated with this single-environment association. First, for programs of even moderate complexity, it takes a significant amount of time to create SMARTSystem databases. In large systems, significant NSE environment creation time has resulted in changed NSE-usage patterns at Sun Microsystems. The time required to generate SMARTSystem databases could easily precipitate similar work-pattern changes for much smaller systems.

Since the costs of creating NSE environments and SMARTSystem databases can be addressed by only infrequently creating new environments (i.e., reusing NSE environments and SMART databases), a more serious issue is that of secondary storage consumption. SMARTSystem databases for relatively small systems (i.e., fewer than 1200 C source lines) routinely

generate databases larger than 20 megabytes. The single-target environment structure depicted in Figure A-2 mitigates this problem to a small degree.

Note that this fast-and-loose use of TFS is justified because there are questions involved with any tighter coupling of the SMARTSystem data files and the NSE component model:

1. The SMARTSystem database and workspaces evolve through creation of files and directories for derived object management, and through communication with external compilers and linkers. Can such transparently (to the SMARTSystem user) created file system objects be managed by NSE? That is, do we need to add all newly created file objects to the NSE component hierarchy? How would the NSE component model be kept up-to-date? Would these strategies work under future releases of SMARTSystem?

2. Can SMARTSystem databases and workspaces be managed in the context of the NSE acquire-reconcile-merge optimistic transaction process model? One feature lacking from SMARTSystem is the concept of "diff" on a database, which is needed to support database merge operations. Note that SMARTSystem does provide a *changedFiles property*[28] for *diff* purposes, but in this case the *diff* is between a workspace and its parent database, not between the parent database and the external repository.

The nested workspace concept is simple, and takes advantage of the distinction between the NSE of the CM data model and the CM data management mechanisms.

Scoped SMARTSystem databases

The advantage of sub-workspaces is the relatively clean coupling of a SMARTSystem database with the NSE managed source files from which the database was derived. The disadvantage is the potential need for many SMARTSystem databases.

An alternative model of interaction between SMARTSystem and NSE workspaces is to associate SMARTSystem databases with NSE *scopes.*[29] A scope is an environment and all of its descendant environments. A scoped database allows only developers within a scope access to the SMARTSystem database.

In order to associate a database with multiple environments, the database itself must be managed outside of the control point defined by the root of the scope. One way to do this is to have the database managed under *NFS,*[30] and simply refer to it from within activated environments. However, it was decided earlier that this solution would be too dangerous since there would be no managed relationship between the SMARTSystem databases and NSE per-process TFS file system context.

NSE does, however, provide a primitive for associating objects with a family of environments (not necessarily restricted to scopes): the execset. NSE provides the execset to support tool

---

28. A *property* is an attribute and associated value maintained by the SMARTSystem object management system.

29. Scope is not an NSE concept; the idea of workspace scopes was discussed in Section A.3.3.3.

30. *NFS*—the Sun Network File System.

families which are dependent upon host/target architecture parameters. In the scoped workspace model, SMARTSystem *databases* are managed within NSE execsets, while SMARTSystem *workspaces* are still managed by individual environments. This approach is illustrated

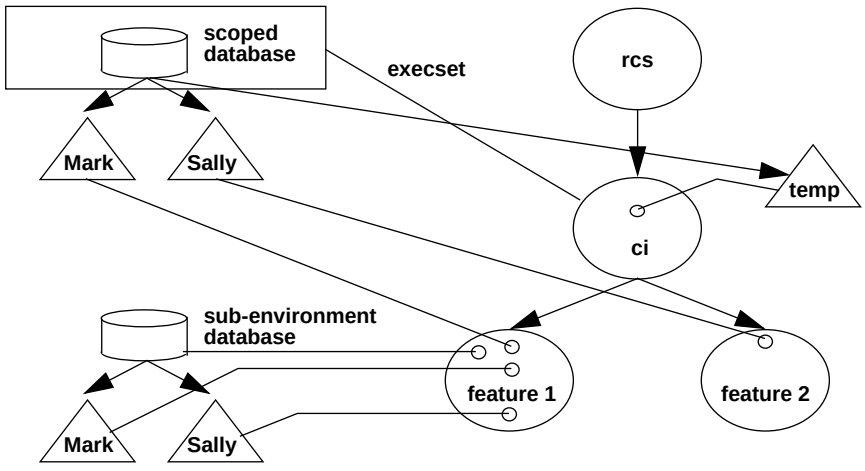

Figure A-7 Scoped SMARTSystem Databases

in Figure A-7. Since child environments by default inherit the parent environment's execset (if the environment is created by the NSE *acquire* command), such a database would be accessible to a restricted set of workspaces.

At first sight this might appear to be a dangerous concept. Since each environment has a unique copy of its acquired components (i.e., configurations) it would appear that scoped workspaces could easily become populated with objects from several environments (through export from the scoped SMARTSystem database into various NSE environments), creating a configuration in SMARTSystem that could not easily be duplicated in NSE. This situation would appear to be only marginally better than having the SMARTSystem database under NFS. However, under certain restricted circumstances, scoped databases can be quite useful. For example, both selective re-synchronization and inter-sibling environment change propagation without the need for an intermediate NSE *reconcile* are attempts at using SMARTSystem to add flexibility to the NSE model of change propagation.

Selective re-synchronization may be desirable if a child environment wishes to incorporate a limited set of logical changes from a parent environment. If these changes are logically distinct but interacting, as might result from several reconciles from sibling environments where each sibling was introducing a new feature, selective re-synchronization could be useful. One way to achieve selective re-synchronization is for both the SMARTSystem database and workspace files to be stored in an execset. If the workspace files existed only in child environments there would be no way to simultaneously have access to workspace files and the reconciled changes. An alternative is to have a conduit SMARTSystem workspace (called "temp" in Fig-

ure A-7) exist for the purposes of propagating the contents from one NSE environment to a family of scoped SMARTSystem workspaces via the scoped SMARTSystem database.

Inter-sibling change propagation can be useful when two developers wish to share changes without propagating these changes to a more global scope. This need might arise if, for example, the cost of propagating the change through NSE would potentially outweigh the benefits. Such would be the case if the propagated changes were experimental, or if the components being reconciled were extremely large, or if the reconcile would cause propagation of too many changes to the parent environment.

Figure A-7 illustrates a hybrid use of scoped and sub-databases. Developers Mark and Sally share a nested database in the “feature 1” environment, while sharing changes between the “feature 1” and “feature 2” environments through the scoped database residing in the “ci” environment’s execset.

There is, unfortunately, an implementation problem in the use of scoped SMARTSystem databases. The problem lies in the NSE implementation of execsets. Since the purpose of execsets is to provide a stable reference point for tools shared by many environment variants, the NSE implements the execset directories as read-only file system mounts. Thus, the database files in the execset are not writable by SMARTSystem clients.

Through the *nseexecset prefix* command, it is possible to determine where the actual database files are stored (recall that file systems are mounted from NSE’s repository, which is distinct from the logical name-space presented via execsets and control points). This seems inelegant, however. Additionally, TFS implements a write-through buffering scheme, so that changes to a mounted file system are immediately visible to the underlying NSE repository, but changes made to the repository will not be visible to the relevant environments until they are re-activated. A cleaner solution than this would be to manage the SMARTSystem databases under NFS, and include in the execset a file which provides a pointer to the appropriate SMARTSystem database directory.

A.4 Remaining architectural aspects of SMART/NSE

The most significant architectural result revealed by this integration experiment concerns the integration of client/server tools with CM, and the possibility of distinguishing classes of services that should be provided by clients and servers. Figure A-8 illustrates the logical view of the sub-workspace integration concept.

In this logical view, the NSE environments communicate with the SMARTSystem database servers. This presents a pleasing architectural view of nested repositories. However, the SMARTSystem tool architecture does not provide a services interface to the SMARTSystem database server. As a result of this opaque-server implementation, a less sophisticated implementation of nested workspace communication is needed. This less sophisticated approach is illustrated in Figure A-9.

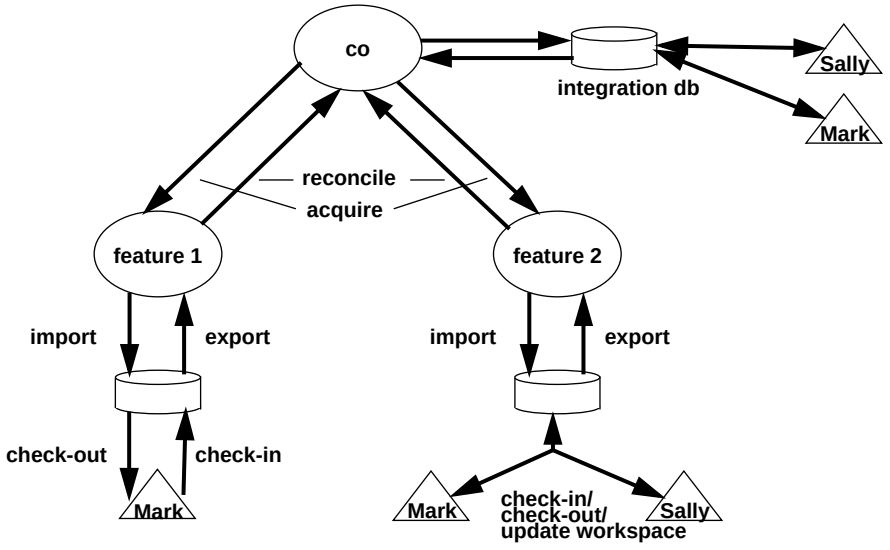

Figure A-8 Workspace Data Flow (Logical View)

Two key points of the implementation of sub-workstations are illustrated in Figure A-9.

1. With respect to the opaque SMARTSystem database server, note that whereas in Figure A-8 NSE workspaces communicated with SMARTSystem database servers, in Figure A-9 the communication is between SMARTSystem

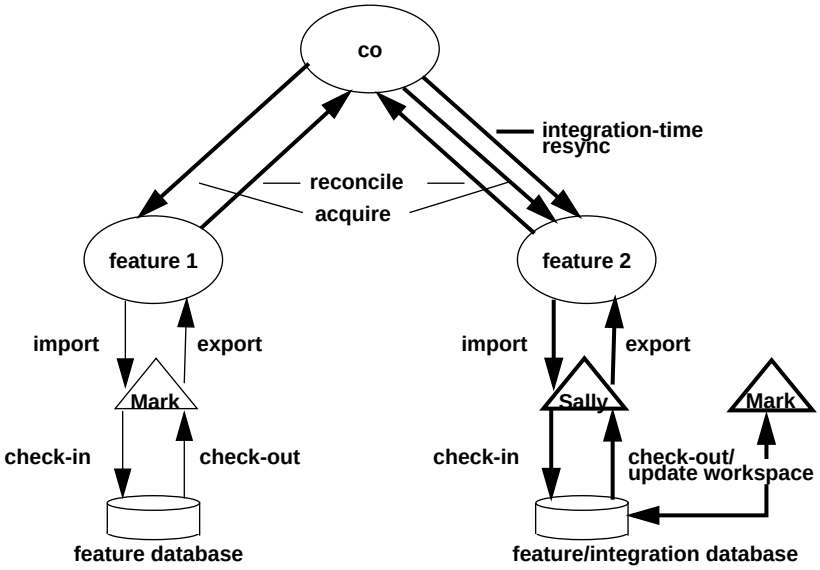

Figure A-9 Workspace Data Flow (Implementation View)

clients and NSE environments. Thus, one of the workspaces (Sally or Mark's) must be designated as the NSE workspace interface; updates to the SMART-System database must be channeled through this designated SMARTSystem workspace. Similarly, exports to NSE must be channeled through some designated environment (although in the case of export it doesn't matter which environment is used).

2. It is possible that a number of server-class CM services could be identified which would facilitate developer-CM integration. For example, the SMARTSystem database server could provide services to notify clients of events. The client SMARTScreen code could be extended to recognize and respond to a selected set of these events. However, lacking such server interfaces, the SMARTSystem/NSE integration must resort to some manual user intervention to keep SMARTSystem databases consistent with the NSE repository. Figure A-9 illustrates the large cost in disk-resources consumed by SMARTSystem databases. In Figure A-8 a SMARTSystem database resided in the NSE integration environment. However, to conserve disk space, one of the SMARTSystem databases of a child environment is re-used as an integration environment. This is where the need for an integration-time *resync*[31] arises.

A.5 Summary of experiment

The experimental integration of SMARTSystem with NSE was fruitful in a number of ways. First, it illustrates the degree to which well-defined process scenarios are needed to provide concrete requirements for making integration design decisions. Second, the experiment illustrates the great latitude available to the tool integrator to integrate tools without having recourse to system or tool source code. Indeed, a running joke during the integration experiments was that we were exploring "how to succeed in integration without really coding." Third, the experiment revealed the need to consider the services provided by tools from a more abstract level than that provided by the implementation itself. The discussion of workspace semantics in Section A.3.3.3 reflects this. Finally, a healthy degree of pragmatism is necessary to avoid expensive integration designs (in terms of system resource consumption and time and effort to effect the integration). One ingredient of this pragmatism is, unfortunately, a detailed knowledge of some (possibly undocumented) structure and implementation details of the tools being implemented.

31. *Resync* is an NSE command for bringing out-of-date components in child transactions up-to-date with respect to components in the parent transaction. Resync is part of the NSE model of optimistic transaction and merge.

Appendix B Synopsis of CM in SMARTSystem

This appendix provides a high-level synopsis of version and configuration management in SMARTSystem.[32] SMARTSystem supports versioning of configurations and workspaces in the following way:

- Versioning in the database is sequential with no branching.
- The entire database is a versioned entity, i.e., composition is not supported.
- Workspaces contain copies of the database; check-in from a workspace creates a new database version identical to the workspace doing the check-in.
- The database does not support parallel (i.e., overlapping) changes. There are two kinds of locks—public and private—either one of which must be obtained from the database in order to modify objects. Objects in the database (i.e., source code) are immutable until a lock is obtained.
  - Public locks of varying degrees of granularity serialize changes; the database ensures that no two clients own a public lock of overlapping regions. Only regions "publicly" locked may be checked-in.
  - Temporary locks may be obtained in a workspace, but changes made in temporary regions are lost unless the region's temporary lock can be upgraded to a public lock. There is no way to merge temporarily locked regions with other database versions.
- There are two other kinds of locks: build-info and check-in locks:
  - Build-info locks are used to modify information related to system builds. This information includes: parse parameters on individual files, link parameters for programs, and the source and text files (i.e., ".c" and ".h" files) included in the database. There is, therefore, some relationship between build-info and system-model.
  - Check-in locks must be obtained in order to create a new database version. They are usually obtained automatically upon check-in, but can be obtained separately in order to lock the entire database. This is useful during a merge operation in a workspace (equivalent to NSE "resync").
- There is no support for merging, per se. The SMARTSystem "merge" concept is equivalent to the NSE "resync," but does not preserve overlapping changes. Merge can be used to:
  - Update a workspace that is out of date with changes made to the database by other SMARTSystem clients. Publicly locked regions in the out-of-date workspace are preserved; since SMARTSystem guarantees non-overlapping locks, no manual source merging is needed.
  - Switch to a different (earlier) version of the database. Merging with earlier database versions is degenerate since SMARTSystem does not allow such merges if the workspace is holding public locks. Such a merge is merely a context switch to an earlier version.

---

[32] This synopsis describes version 1.6 of SMARTSystem.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S
CMU/SEI-92-TR-5

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESD-TR-92-5

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (City, State and ZIP Code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (City, State and ZIP Code)
ESD/AVS
Hanscom Air Force Base, MA 01731

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESD/AVS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (City, State and ZIP Code)
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Issues and Techniques of CASE Integration with Configuration Management

12. PERSONAL AUTHOR(S)
Kurt C. Wallnau

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM TO	March 1992	82

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse of necessary and identify by block number)
CASE tools, Environments, Software Development, Computer-Aided Technology

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Commercial computer-aided software engineering (CASE) tool technology has emerged as an important component of practical software development environments. Issues of CASE tool integration have received heightened attention in recent years, with various commercial products and technical approaches promising to make inroads into this difficult problem. One aspect of CASE integration that has not been adequately addressed is the integration of CASE tools with configuration management (CM)—including both CM policies and systems. Organizations need to address how to make CASE tools from different vendors work effectively with an organization's CM policies

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT ☐ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
John S. Herman, Capt, USAF	(412) 268-7631	ESD/AVS (SEI)

and tools (in effect, integrate CASE with CM) within the context of the rapidly evolving state of commercial integration technology. This report describes key issues of the integration of CASE with CM from a third-party integrator's perspective, i.e., how to approach the integration of CASE and CM in such a way as to not require fundamental changes to the implementation of the tools or CM systems themselves.