# The STCL test tools architecture

by C. Williams
H. Sluiman
D. Pitcher
M. Slavescu
J. Spratley
M. Brodhun
J. McLean
C. Rankin
K. Rosengren

***The Software Test Community Leaders (STCL) group is an IBM-wide initiative focused on improving software test and quality practices within the corporation. In 1999, we began working to develop an architecture to integrate both new and existing test tools into solutions for use in the testing organizations across IBM. This paper discusses the requirements for the architecture, as well as the issues associated with developing a solution architecture for a large base of tools that span a variety of platforms and domains. The architecture is being designed and developed to address three concerns for integrating testing tools: integration of the data across tools and repositories, integration of the control across tools, and integration to provide a single graphical user interface into the tool set. Because of the heterogeneous nature of the platforms and domains the architecture must support, extensibility is essential. We address each of these three integration concerns using an open-source framework that operates on a set of standardized but extensible entities.***

In 1998, IBM formed the Software Test Community Leaders (STCL) group to address issues associated with software testing and quality. The group consists of key technical professionals and managers from testing groups across the various divisions of IBM. They collectively represent the depth and breadth of software testing expertise available within the corporation. One of the first tasks the technical team undertook was to identify and categorize leading tools and practices in use across the divisions, with the idea that this would enable key tools and practices to be shared. As a result of this exercise, the team produced a list of best practices with supporting tools and discovered two problems that would inhibit tool and practice sharing.

1. Different labs within IBM had different tools supporting the same practice. A variety of manual, STCL supported, and third-party or homegrown tools are used across labs within IBM.
2. Testing groups were mixing manual processes with both STCL recommended and homegrown and third-party developed tools. Because these tools were not designed to work together, human intervention or custom integration was required (Figure 1).

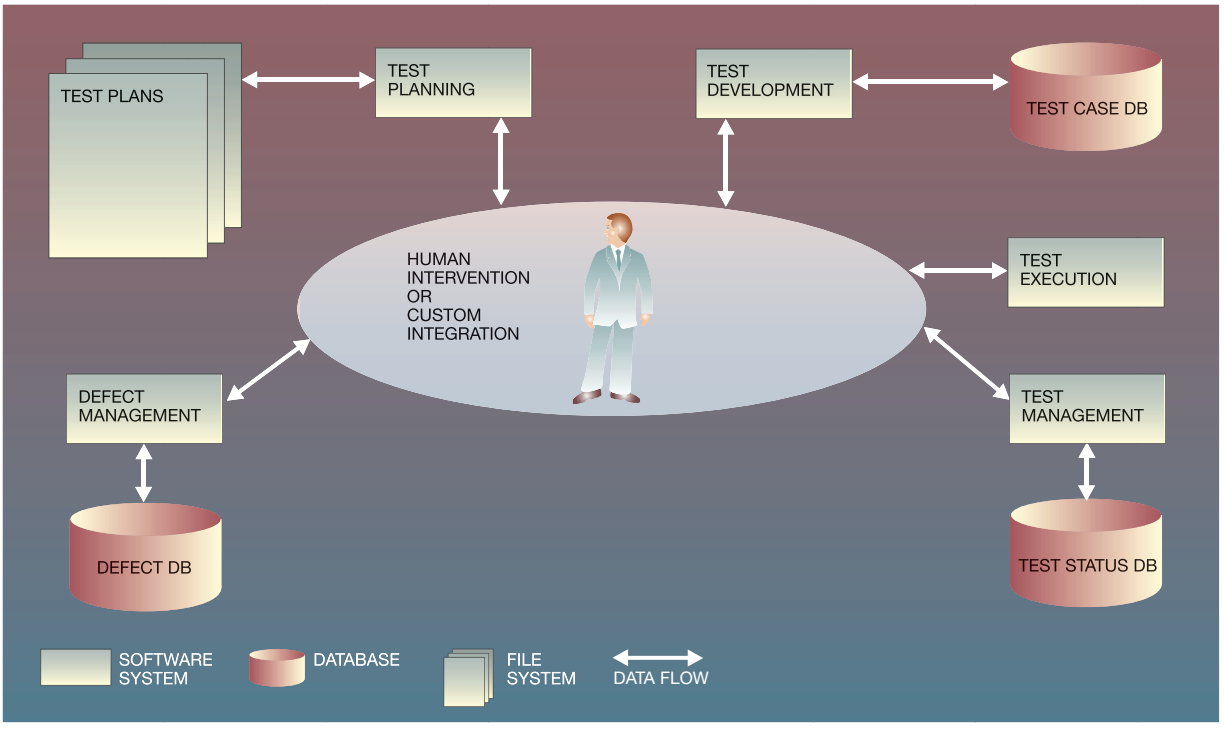
The multitude of tools with similar functionality within IBM is due to the variety of platforms and products the corporation develops and supports. Also, different labs across IBM have variations in their testing processes. A process is composed of one or more testing practices. While the practices might be basically the same, the processes that the practices support can be markedly different from lab to lab. This leads to the development of tools customized for site-specific processes, even though they are based on practices that are common throughout the corporation.

In mid-1999, the STCL technical leaders decided to take an architectural approach to addressing these two problems. The STCL Architecture Committee was formed from a subset of the technical leaders who represented the various labs and were interested in developing an architecture. The committee started its work by exploring high-level solutions that de-




0018-8670/02/$5.00 © 2002 IBM

Figure 1 A sample of heterogeneous and disconnected test tools




scribed how tools might interoperate when supported by an architecture. Figure 2 shows one example of such a solution. The group developed other views as well, with the ultimate goal being an architecture that would provide support for customizable solutions.

In the remainder of the paper, we cover the following topics. First, we discuss the requirements that the architecture must satisfy. Then we present the architecture, focusing on three concerns: data, control, and GUI (graphical user interface) integration. Next we discuss how new tools can use the architecture, how legacy tools can be integrated, and how the architecture meets the requirements. Finally, we present conclusions and future work.

## The requirements

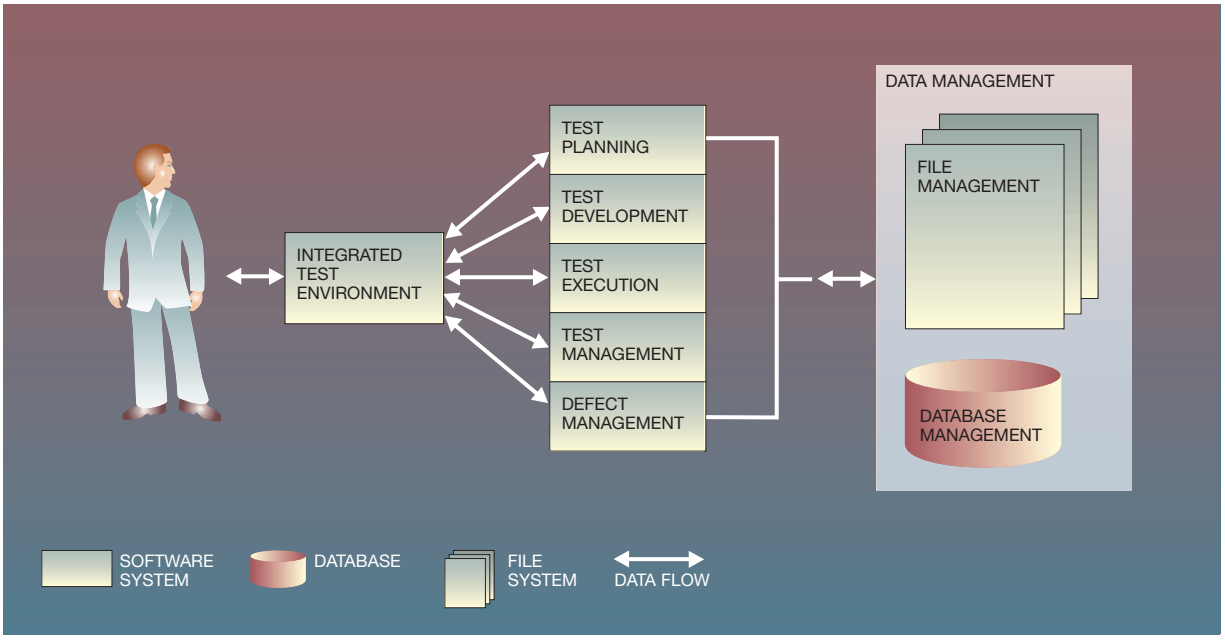
At the highest level, there are three requirements that the architecture must meet:

1. It must provide a mechanism for integrating the variety of tools recommended by the STCL.
2. It must support site-specific tools and testing processes.
3. It must support legacy tools, while also providing a road map for new tool development within IBM.

The problem of integrating heterogeneous, independently developed tools has received attention from other parties, both inside and outside IBM.[1,2] The integration is usually accomplished using a three-phase approach, which includes data integration, control integration, and interface integration. This is the approach that the STCL architecture committee selected, and it led to a more detailed set of integration requirements:

1a. Data integration must make test-related data available in an open manner regardless of the tool or repository in which the data are stored.
1b. Data integration must provide a way to maintain associations among related data, even if the data reside in different repositories.
1c. Control integration should support invocation of externally available functionality on tools that comply with the architecture. This will be impor-

Figure 2 High-level view of an architecture-supported testing solution




tant for building highly automated testing environments.

1d. GUI integration will result in a single user interface for accessing all of the architecturally compliant tools.

1e. GUI integration should not preclude using the tools as they are currently used today. This is important because different groups will migrate to the new architecture as business conditions permit.

In order to meet requirement 2 (support for site-specific tools and processes), we developed the following set of more detailed requirements:

2a. A "plug-and-play" mechanism for incorporating site-specific tools must be part of the architecture. This mechanism should be easy for tool owners to adopt.

2b. Control integration should allow workflow customization to support variations in the testing process.

By "plug-and-play," we mean that a tool that has complied with the architecture should be able to be incorporated by a testing group into an architecture-based solution with minimal effort. Furthermore, this solution should support customization of processes using a workflow definition language.

Requirement 3 addresses two needs: to support and protect the existing investment in legacy tools, and to provide a road map so that new tools are designed to interoperate from the start. The following detailed requirements support these high-level goals:

3a. Enhancements to legacy tools must be localized, with no changes to internal tool control structures or data representations.

3b. The architecture should provide guidance for the design of new tools in the form of standardized application programming interfaces (APIs), standardized testing entities, and other reusable items.

3c. Both legacy and new tools should be supported using reusable components.

3d. The legacy and new tool requirements should be met with as little variation within the architecture as possible.

These requirements form the foundation for the development of the architecture. Given the size of IBM, converging on a set of requirements is a difficult and ongoing challenge. To help with this task, our pro-

cess involves pilot installations of beta technology to ensure that the solutions built using the architecture meet the needs of real testers within IBM. We also check with tool owners to determine whether or not the architectural components are easy to use and well documented.

## The architecture

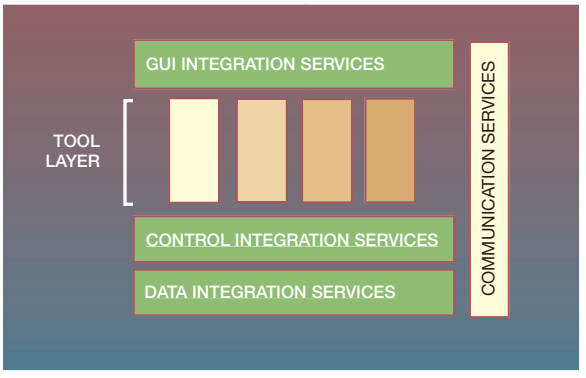
The STCL architecture addresses three types of tool integration: data integration, control integration, and GUI integration. With data integration we attempt to provide a unified view of the data within the testing organization, regardless of where the data are stored. We also address associations among related data artifacts that might be stored in separate repositories. Control integration allows unrelated tools to invoke behavior on one another in a generic way. For example, a tool invoking behavior on another tool does not need to know about the API of that tool. With GUI integration we attempt to provide a single, consistent interface for all of the architecturally enabled tools.

Although the architecture specifies three levels of integration, we recognize that not all levels are appropriate for all tools. For example, certain execution tools do not require a GUI, and hence would not be part of the GUI integration level. The ability to support varying levels of integration is implemented by using a hierarchical layering approach.[3] We discuss this approach in detail in this section. The defined levels of integration for the architecture are:

1. Data exchange. The tool supports the ability to read and write data in the data-exchange format of the architecture.
2. Data integration. The tool supports standard functionality on its artifacts (create, read, update, delete) as well as associations with artifacts within other tools.
3. Control integration. The tool provides the ability to invoke operations on it via the generic control integration services.
4. GUI integration. The tool supports an interface within the standard GUI environment.

**The high-level layered architecture.** The three levels of integration are supported by four basic architectural components: the GUI integration component, the control integration component, the data integration component, and the communication component. Each component supplies a set of services that can be used by a tool. The GUI integration component provides a set of common GUI services. The control integration component offers services for operation invocation and automation, while the data integration component manages data content and associations. The communication component connects the tools to the GUI, control, and data integration components.

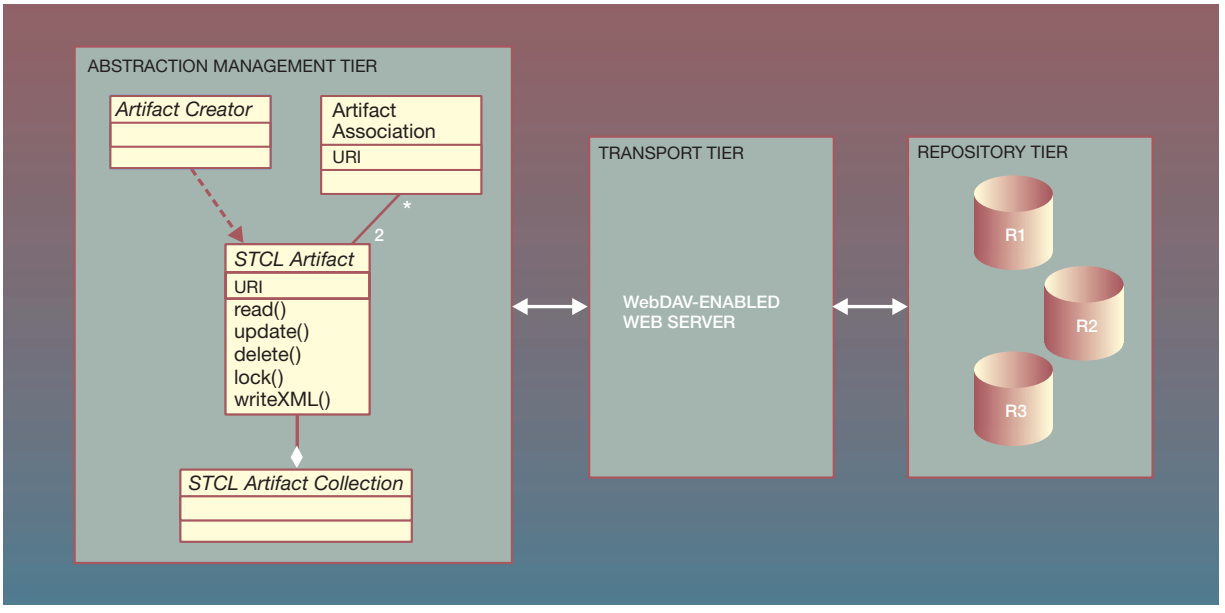
Figure 3 The modified layered architecture




The approach of using components that provide a set of reusable services is similar to the NIST/ECMA Reference Model for Frameworks of Software Engineering Environments, a model that provides a common foundation for software engineering environments.[4,5] This standard defines a set of service groupings that support the three levels of integration just described. These service groupings are a superset of the above components and can be combined in a precise way to specify an architecture.

At the highest level, the STCL architecture is a modified layered architecture (Figure 3). It is modified because an architecturally compliant tool may use any of the services in the three integration components. Tools within the tool layer may communicate, directly or by using the communications component, with either adjacent layer (GUI integration and control integration) and via the communications component with the data integration component. In contrast, the GUI, control, and data integration components obey a strict layering constraint and can initiate communication with only the layers directly adjacent to them. In Figure 3, the tools following the strict layered model are shown in green. This means that the GUI layer initiates communication only with the tool layer, the control integration layer initiates communication only with the tool or data integration layers, and the data integration layer initiates

Figure 4 Three-tier logical architecture for data integration




communication only with the control integration layer. This layering structure makes modifying and maintaining the integration components easier by limiting the interfaces that must be maintained among them.

Having shown a high-level view of the architecture, we now discuss the data integration, control integration, and GUI integration components in more detail. The communications component, based on HyperText Transfer Protocol (HTTP) and interprocess communication (IPC), is straightforward and does not require a detailed discussion. We present the three components using the logical, process, and physical views of the architecture, where appropriate. These views are part of the "4 + 1" approach to describing software architectures.[6] The logical view shows how components are structured from a conceptual point of view. Process views specify the thread and process structure that provides the dynamic behavior of the components in the logical view. Physical views show how the various processes are allocated to hardware resources in groups using the architecture. The development view is implementation-specific and is not discussed here.

**The data integration component.** The data integration component provides a coherent way to access and manage all of the test-related data within an organization. If there were no legacy tools, and if data artifacts were not constantly being defined and refined, data integration would be straightforward. This is not the case. Legacy tools often contain vast amounts of data that need to be shared and integrated with data in other tools. New artifacts are a necessity, and artifact structures change as product teams change their focus. The data integration component must take into account two factors: (1) The data in the organization are distributed within a heterogeneous set of repositories and tools; (2) the data artifacts need to be flexible and extensible. Furthermore, associations among related data should be captured and preserved, even when the data reside in different repositories.

*Data integration: Logical view.* Logically, the data integration component reflects a three-tier architecture, where the first tier is the data abstraction tier, the second tier is a server for accessing the heterogeneous data repositories and tools available within an organization, and the third tier is the data repository tier containing these tools and repositories. This is shown in Figure 4.

Data abstraction is the ability to represent and operate on all data in a uniform way, regardless of

where the data reside. To support this capability, the data abstraction tier serves two purposes. First, it offers support for creating and managing the various artifact types, wherever they reside. Second, it provides management services to maintain associations among artifacts, even when they are stored in separate repositories. To support data abstraction, we developed a representation of a data artifact that standardizes the basic set of operations that an artifact must support. These include create, read, update, and delete operations as well as locking and writing the artifact out as a stream in the standard data resolution format. When combined with the data resolution and data transport capabilities, data abstraction is a powerful mechanism for uniformly representing and managing data.

In addition to the generic STCL artifact representation, the STCL Architecture Committee has defined a draft set of standard, extensible representations for common test-related artifacts including test cases, test suites, defects, and testing-related host configuration information. As part of this process, the committee plans to join with representatives from other organizations to discuss standardizing these testing artifacts. Because these artifact definitions will be developed based on feedback from representatives across IBM and other interested parties, we believe they will be widely applicable. Other industries have developed similar standards, such as FpML** (Financial products Markup Language) for financial products.[7] Standards for test management artifacts could be a very potent force for better software development and testing tools.

The transport tier addresses the issue of sharing the data between the abstraction tier and repositories within the architecture. Because we may have several repositories running on several separate machines, the transport mechanism must support networked data sources. It must also support a simple, open structure for accessing data that can be mapped to proprietary mechanisms used by particular tools and repositories.

HTTP/WebDAV is a transport mechanism that meets these requirements, and it serves as a basis on which a robust transport structure is constructed. WebDAV stands for Web Data Authoring and Versioning. It is an IETF (Internet Engineering Task Force) standard for reading, writing, and updating resources using the World Wide Web, and it has emerged as a tool for building complex collaborative environments.[8] It has three major advantages: (1) It is in widespread use and is available on all platforms used within IBM; (2) it is easy to understand; and (3) it is an open extension of the popular HTTP standard.

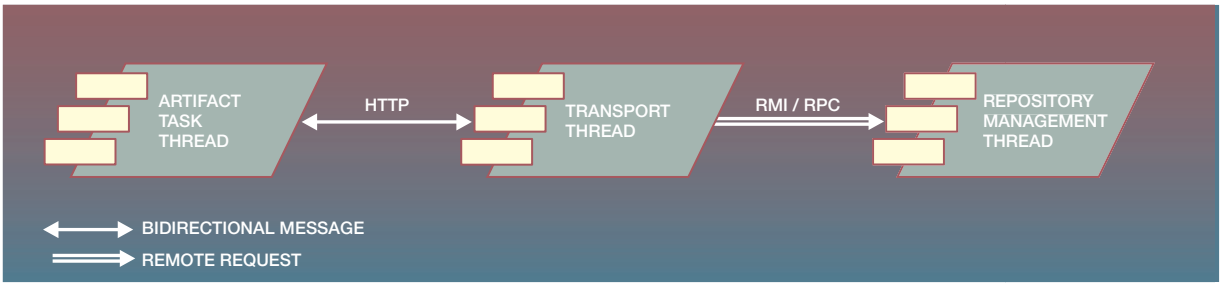
The WebDAV standard basically makes the Web writable by adding several capabilities to HTTP, beyond the ability to obtain data as a stream of HTML (HyperText Markup Language) or XML (Extensible Markup Language). WebDAV works with resources, which can be logically considered as pages on a Web server. WebDAV provides the capability to access, update, lock, write, and delete these resources. They are referenced using a URI, WebDAV commands, and HTTP. (URI stands for Uniform Resource Identifier. URI is the preferred W3C [World Wide Web Consortium] term that subsumes what is also known as a URL.) The transport and repository tiers in the architecture provide a mapping from what appears to be a standard URI location on the Web server into an actual data location in a repository. Each artifact on the abstraction tier has a URI that can be used to update or access it in a shared data repository or tool.

Even with standard representation and transport tiers for the various artifacts, an integral step in obtaining an integrated view of the data is to obtain access to the data within a given repository. For a repository to participate in the architecture, it must allow data access via the WebDAV method. This can be supported directly by the repository, or by using a simple framework that we have developed to support legacy tools. Legacy tool support will be discussed in more detail in a later section.

Each repository must also offer its data in a form that can be understood and managed in the abstraction tier, a capability known as data resolution. In the past, data resolution across the variety of platforms and applications in use within IBM was a significant issue, with proprietary formats being the rule. The advent of the Extensible Markup Language (XML) has greatly simplified the question of data resolution. Each artifact defined in the architecture can have an associated schema defined using XML. Tools that store and provide access to artifacts simply obtain the data and format the data into well-formed XML as defined by the schema. Tools using the data receive and parse the XML, which is straightforward given the variety of XML parsers available.

*Data integration: Process view.* For a given artifact that has persistent residence in a single repository,

Figure 5 Process structure of the data integration layer




three threads are relevant: the thread of control relating to the artifact itself within the abstraction management system, the thread of control that handles transport within the Web server, and the thread of control in the back-end repository system. These are captured in Figure 5.

We use threads in this discussion because many components have been realized using the Java** language. If similar components are realized in other languages, processes can be substituted if necessary. The basic processing paradigm is as follows:

1. A tool requests an operation on an artifact in the artifact abstraction layer. Certain operations (such as read) return the information directly if the artifact is already present in the layer. Others require further processing.
2. If the operation requires processing involving either a read from or an update to the repository layer, the artifact is locked and the appropriate command is issued to the transport thread.
3. The repository management thread fulfills the request and returns any data to the transport thread, which returns it to the artifact task thread.

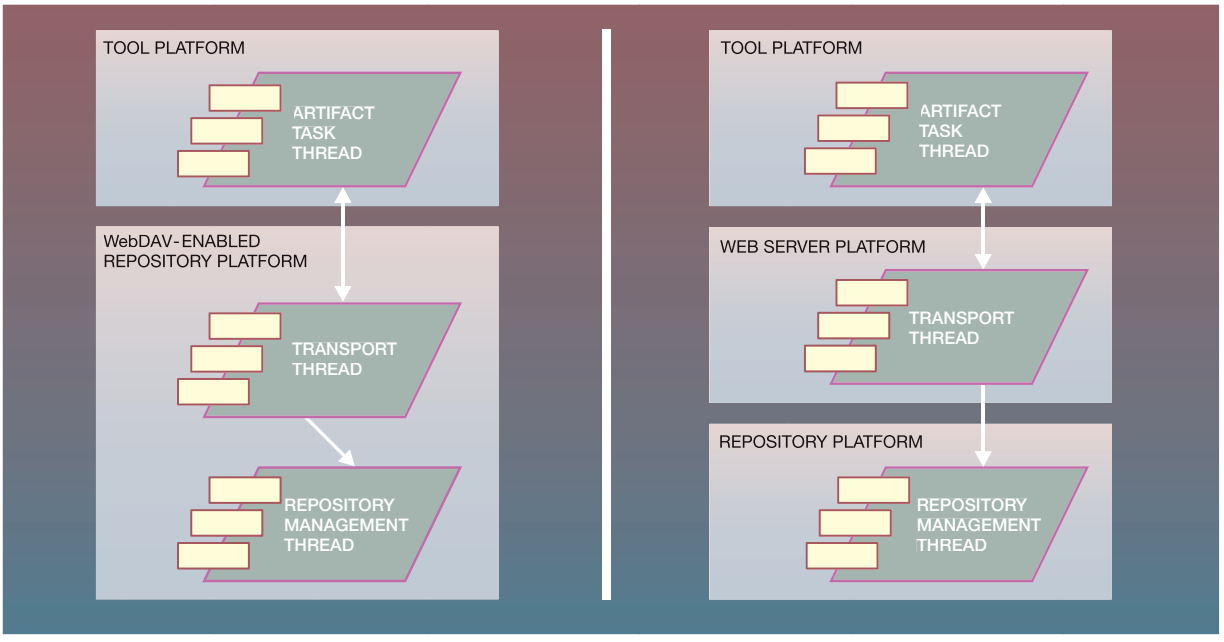
We must address synchronization because several artifacts can be present within the abstraction layer, and they can be accessed by multiple tools at any given time. When a tool updates an artifact in the abstraction tier, the following sequence of events occurs:

1. The update operation requests a lock on the artifact. The artifact issues a WebDAV lock command via HTTP to the transport thread.
2. The transport thread forwards the locking message to the appropriate repository management thread, which obtains a repository lock or a failure message.
3. If the repository lock was successful, the artifact is locked in the abstraction layer and the operation proceeds. When the update operation is complete, both locks are released.
4. If the lock could not be obtained, the operation returns a message to the requester.

*Data integration: Physical view.* There are several possibilities for the physical layout of the data integration portion of the system. Figure 6 shows two common views. On the left is the enabled repository view. The actual thread structure might differ from that shown, because the repository developer can customize the implementation. On the right is the nonenabled repository view, in which the WebDAV requests must be handled by a Web server, which will typically reside on its own hardware platform. The repository runs on a third piece of hardware, and is accessed via RMI/RPC (Remote Method Invocation/Remote Procedure Call) as shown in the process view. In both views, the artifact task thread is shown physically running on the tool platform. This is because the artifact abstraction management tier is provided via an included library. There are many other possible configurations for the physical architecture, including configurations that provide redundancy and fault tolerance by duplicating components. A group can choose which of these to use, depending on its needs and resources.

**The control integration component.** Control integration confers the ability to distribute and customize the flow of control among the various tools that are integrated in the architecture. A flexible, customizable control integration mechanism is essential to allow support of the multitude of testing processes

Figure 6 WebDAV enabled (left) and nonenabled (right) physical views




used across the labs. This is especially important to support the creation of an integrated test environment (ITE), which is analogous to the integrated development environments (IDEs) that developers use, but focused on testing. ITEs and IDEs have a significant overlap in functionality, because testing projects usually involve code development. However, testing also addresses executing the software system under test to determine whether or not it meets its requirements. Execution is a complex and time-consuming operation, because it involves steps beyond simply running a test case. These steps include setting up the environment, executing the test, recording the outcome, cleaning up after the test is complete, and recording failure information. Further complicating matters is the fact that most of these steps are quite specific to the product being tested, the execution environment, and the processes used by the testing organization. It is especially important that the control integration strategy offer support for multiple platforms and processes.

At a high level, a control integration strategy must support the following capabilities:

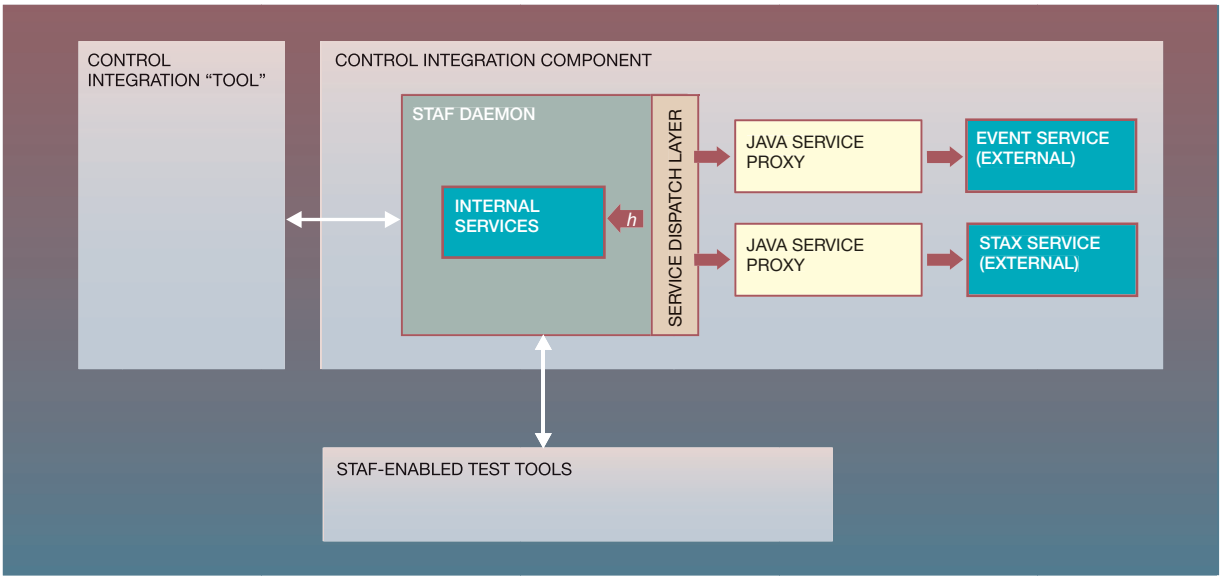
1. It should define a tool- and environment-neutral mechanism for coordinating control among the tools used in the testing process.
2. It must provide a tool- and environment-independent way to specify and enforce a software testing process.

We address these goals using two types of control integration: event management and workflow process control. Event management is essential for control integration. It allows resource providers within the architecture to be made aware of asynchronous events when they occur, and they in turn respond by performing actions specified in their event handlers.

Workflow process control is more complex. It involves coordinating and enforcing control and information flow among several distributed applications, and it is essential for supporting varied testing processes.

The Software Testing Automation Framework (STAF) offers both of these control integration facilities, as well as many others. STAF is a highly portable framework that provides reusable services for automating testing tasks in a peer-to-peer manner. Here we present only the services that are relevant to the control integration architecture. STAF also supports many other services including queues, trust services, semaphores, process control, logging, etc.; more information on STAF is available in a separate

Figure 7 The logical structure of the control integration architecture




paper.[9] Additionally, the framework is extensible, meaning that groups and tool owners can write other services as needed.

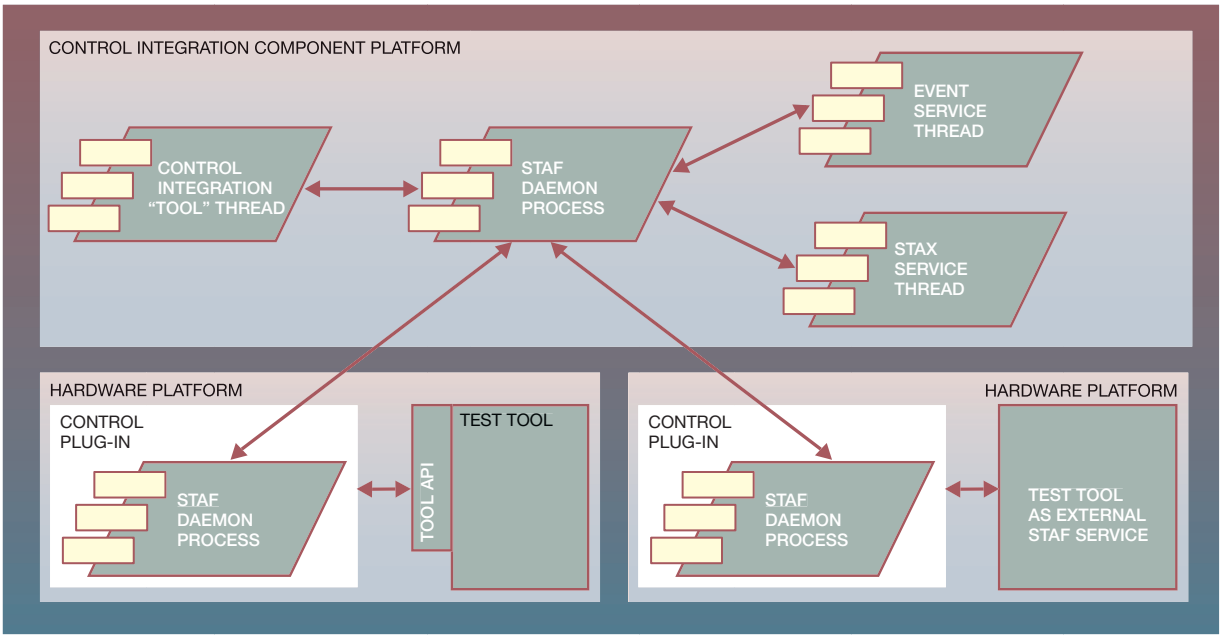
STAF's event service provides the facilities for implicit invocation. To use the event service, a particular machine is typically designated as the event server. All tools interested in receiving information about a particular event register with this server, and other systems generate events to the server in order to invoke behavior on all registered tools. In addition to registration and event generation, the service also supports mechanisms to acknowledge events, query events, and "unregister" for events. To support control integration across tools, we are currently refining a set of standard but extensible events (analogous to the standardized artifacts in the data integration component). This set includes invocation events (such as starting a test harness), notification events (such as software build notification, test completed notification, and problem notification), exceptional events, and others.

STAF also provides the STAX (STAF eXecution engine) workflow manager as a service. This service accepts workflow definitions, which define the workflow that accomplishes particular testing processes. The workflow definition language is very expressive and has the ability to invoke any of the STAF services. It also contains high-level programmatic functionality, including selection, iteration, sequential and parallel grouping, signal handling, and exception handling. STAX works using the event service, and together they form the basis for the control integration component.

*Control integration: Logical architecture.* The logical structure of the control integration architecture is shown in Figure 7. The heart of the logical architecture for control integration is the control integration component. Within this centralized component is a copy of the STAF daemon, as well as the event and STAX services. The control integration component is connected via the control integration "tool" to the GUI component. This allows the GUI component to interact with non-GUI tools in the architecture via STAF events or STAX jobs. Other STAF-enabled tools can also use the control integration component as the hub for control of other tools via events or STAX jobs.

*Control integration: Process and physical architecture.* Figure 8 shows a combined view of the process and physical architecture of the control integration component. Each hardware platform that participates in the control integration infrastructure must have a STAF daemon running on it. To communicate with

Figure 8 The process and physical architecture of the control integration layer



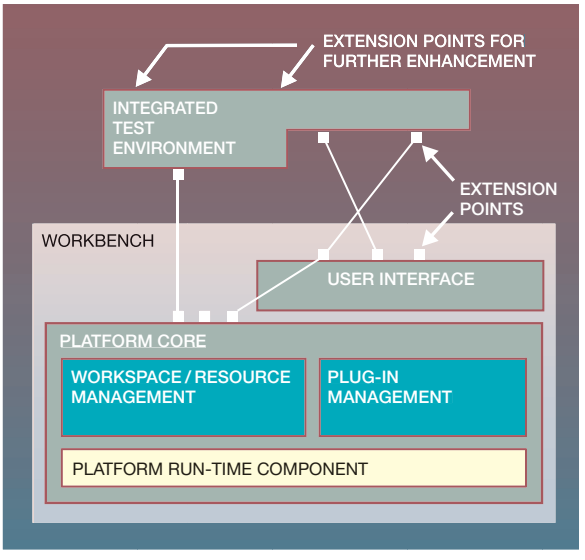

the STAF control integration infrastructure, and to reduce the amount of overhead required across the system, the control integration "tool" thread runs within a Java virtual machine (JVM) on the same hardware platform as the STAF control component. Thus, this is indeed a virtual tool, not directly running in the tool layer, even though it appears to from the perspective of the GUI integration component.

A control integration "tool" thread communicates directly with the STAF daemon running on the same hardware platform. This allows events to be generated or STAX jobs to be initiated from the GUI component. Individual events and STAX jobs all run within separate threads, but within the same JVM. In turn, these services use the STAF daemon to communicate with other tools using the architecture. As the figure shows, two possible modes exist for tools to use the control integration services. One mode allows the tool to communicate with the STAF subsystem via an API on the tool, which means that no STAF commands will appear in the tool's "native" code. The other mode lets the tool invoke and respond to STAF directly within its code, which in essence is the tool acting as an external STAF service. Which of these two modes of integration is used will depend largely on the architecture of the tool.

In addition to the event and STAX workflow services provided by STAF, several other services can be used in the architecture. For example, the semaphore service can be used to synchronize access to resources across multiple hardware platforms within the environment. Also, STAF supports the development of additional services as required. In fact, we are currently exploring the definition of a new set of services that would provide data integration capabilities for any tool that is already STAF-enabled.

**The GUI integration component.** The goal of GUI integration is to provide testers with an interface rich enough to serve as a single point of access for the tools typically used in the testing process. This interface, when combined with the data and control integration capabilities of the architecture, will provide testers with an integrated test environment (ITE) similar to the integrated development environments (IDEs) commonly used in development today. We are currently using the Eclipse-based WebSphere* Studio Workbench (hereafter referred to as "the workbench") to develop a sophisticated desktop environment for use on the testers' workstations. Eclipse is an extensible, open-source platform that provides a common set of components and services for building development tools.[10] The workbench provides

Figure 9 The logical view of the GUI integration components




extension points, which are well-defined locations for functional extension by plug-ins. Plug-ins provide implementations (known as extensions) for these points, thereby enhancing the capabilities of the platform. Plug-ins may define their own extension points, and thus can be extended by other plug-ins. New tools can integrate into the platform by developing plug-ins that use the APIs and extensibility services of the environment to participate in the ITE.

The appeal of an ITE is that the tester has a single interface at which to work, one that leverages existing tools used in the organization. The interface uses the data and control integration infrastructure to present a coherent, unified view of the testing project. Today, testers move from tool to tool as they perform different tasks. For example, a tester might use an execution tool to run tests, a different tool to log the results, and yet a third tool to enter the problems discovered during testing. Not only is this inefficient, but it also fragments the data in the testing organization; hence the need for data integration via the architecture. Developing a common user interface that can be used by various tool types will significantly boost productivity and help testers to realize the gains made possible by data and control integration.

The remainder of this section presents a high-level view of how we are using the workbench to develop an ITE. We do not present an in-depth discussion of the Eclipse project; that is beyond the scope of this paper. Instead, we give an overview of how it supports GUI integration in the architecture and interacts with the other integration components discussed earlier.

*GUI integration: Logical architecture.* The workbench is a user interface (UI) plug-in that runs on top of the Eclipse platform core. The core components include the platform run-time code and two sets of services: plug-in services for integrating new behavior into the workbench and workspace and resource management services for managing projects, folders, and files.

A tool owner using the workbench provides implementations of specific extension points in the workbench or another plug-in. He or she can then use all of the services supported by the workbench. Figure 9 shows the basis structure of the core components, the UI, and a tool implemented as a plug-in to the workbench environment.

The STCL is developing an extensible plug-in that can be used or extended to deliver an ITE to a testing organization. This plug-in will support the set of extensible artifacts discussed in the subsection on data integration. Tool owners may use the artifact definitions and plug-in directly, or they may extend and customize them using built-in extensibility mechanisms.

*GUI integration: Process and physical architecture.* The process and physical architecture of the workbench is straightforward. The core components and tools typically run on the same hardware platform. A discussion of the workbench thread structure is beyond the scope of this paper, other than to note that all plug-in components run within the UI thread(s) of the workbench.

## Application of the architecture

In this section, we explore how the architecture affects the creation of new tools, as well as how legacy tools can be integrated into the architecture. We also use simple examples to explain how the architecture provides the required capabilities identified earlier.

**Building new tools for the architecture.** New tool owners can build tools that are "architecture-ready" by complying with the architecture across three basic areas: artifact representation (data integration),

event representation (control integration), and plug-in representation (GUI integration). These three areas are supported using standard, extensible components and code available within the STCL frameworks library.

If a tool owner wishes to communicate with other tools via data integration, he or she must follow two guidelines during design. First, the tool must use the standard STCL artifacts. This set is extensible, so that tools defining new artifacts not previously available in the STCL can use the data integration services. However, any new artifacts must be extensions of the base STCL artifact. Second, the tool should use the abstraction management tier when creating, reading, updating, or deleting objects. The tier guarantees support for associations between objects that exist in other repositories. Access to this tier is via a set of APIs, which are provided as part of the data integration library.

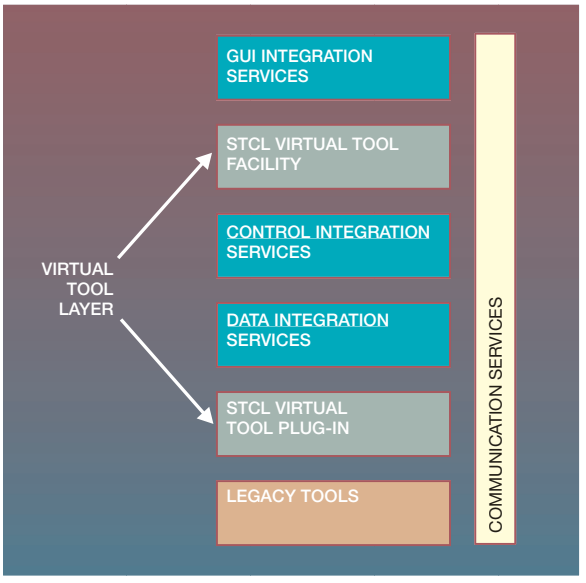
Control integration is supported by building tools that provide facilities for using the STAF event service. This requires that tools use the appropriate standard STCL events, which are defined in the STCL control integration library. Like the artifacts, these events are extensible. The tool must also support the standard STAF API and provide facilities to generate or respond to any relevant events. This requirement can be supported by creating a tool-specific API and a component that maps STAF events to the tool using this API, as well as by building the tool itself as a STAF service.

GUI integration is supported by building the user interface as a workbench-ready plug-in. The STCL GUI integration library is currently under construction, and will consist of a set of plug-ins for developing an ITE. Like the STCL artifact and event sets, these plug-ins will be extensible. They will allow plug-in extensions to interact with the data abstraction tier and control integration layer.

We hope to support the development of architecturally compliant test tools through the use of an STCL test tool development environment (TTDE) built within the workbench. This environment will be related to the standard plug-in development environment (PDE) and will support the development of tools that can be easily plugged into the data, control, and GUI integration frameworks.

**The architecture and legacy tools.** As we noted in requirement 3, the architecture must also allow the use of legacy tools within an ITE solution. New tools developed with an awareness of the architecture would use the services as just described; however, legacy tools are often unable to do this directly without significant modification. To avoid these modifications, legacy tools will use the STCL virtual tool facility (VTF), which we are developing to enable artifact management, control integration, and GUI integration capabilities that are difficult or impossible from the legacy tool directly. The virtual tool plug-in (VTP) maps between the API used by the VTF and the API supported by the legacy tool. Figure 10 shows the VTF within the modified layered architecture. In this configuration, the VTF plays the role of a tool within the tool layer, and the VTP provides connection between the legacy tool and the VTF. We have developed a beta version of the VTF that contains limited functionality and have deployed it in pilot projects. We are currently enhancing this facility based on the feedback of legacy tool owners who have used it.

Figure 10 The modified layered architecture for legacy tools




When it is completed, the VTF will provide the following capabilities:

1. Connection with legacy tools for data integration using the VTP. This functionality provides for the definition and management of artifact types stored in the legacy tool. The VTF implements the

abstraction management tier on behalf of the legacy tool.
2. Support for displaying information in the legacy tool via the workbench UI. The VTF acts as a plug-in for legacy tools, mapping new artifacts and actions into the data and control integration frameworks, as well as into the legacy tool via the VTP.

The VTF/VTP basically acts as a surrogate tool, making the legacy tool appear to be a STAF event-enabled repository.

The VTF communicates with the VTP via the communications component. Typically, there will be a separate Web server acting as a transport layer between the VTF and the VTP. The VTP maps the creation, reading, updating, and deletion of artifacts into the native API of the legacy tool. It also provides a modular component for generating events and receiving event notifications. These events are then converted into actions on the tool's native API.

Legacy tools that make use of the VTF must implement the VTP. The VTP requires that the tool have a well-defined API that can handle the creation, reading, updating, and deletion of the artifacts within the legacy tool. If the tool is going to participate in control integration, the API must also provide the required functionality to respond to or generate the events of interest. Because data and control integration requests can come into the legacy tool from multiple clients, the tool must support server-like behavior. Fortunately, most of the legacy tools within IBM that are currently under consideration for integration with the STCL architecture meet these requirements.

**The requirements revisited.** We identified three main requirements for the architecture. It should provide a way to integrate the various STCL and other tools, support easy installation and customization, and support legacy tools. Each of these gave rise to several subrequirements, which are now discussed.

*Integration of STCL tools.* Subrequirements on the integration of STCL tools include: the capability to access data regardless of which repository the data reside in; maintenance of associations among data across repositories; ability to invoke functionality on tools within the architecture; possessing a single GUI for tester access; and the ability to use legacy tools in the manner available today. The following scenario illustrates how these requirements are supported in the architecture:

1. A tester defines a new test case artifact in the integrated GUI, which is provided as a plug-in to the workbench by the tool that manages the artifact. This covers the single GUI for tester access requirement.
2. When submitted, the test case is created on the abstraction management tier and is associated with any other artifacts specified during its creation. Association specification may be either explicit or implicit (based on the context of the creation). This fulfills the requirement that associations be maintained across repositories.
3. A permanent version of the test case is created in a test case repository, using the data integration WebDAV capabilities. The correct repository is determined by site-specific mappings on the artifact management tier. This artifact will be permanently accessible to any tool using the data integration component by retrieving it using the URI established when it is created. This is an example of access across repositories.
4. The tester may also invoke an execution tool to run the test case from the GUI. The execution tool is connected to the GUI via a control integration plug-in in the tool layer, which generates the appropriate STAF event to invoke the tool. This is an example of generic invocation using STAF. Note that GUI-based tools may also provide direct access to their functionality via API calls from a tool-specific plug-in in the workbench.

Finally, we required that legacy tools be usable in their current manner until it is possible to migrate to the architectural version. The VTP that makes a legacy tool compliant with the architecture does nothing to change the tool, so it can continue to be used outside of the ITE solution when necessary.

*Ease of installation and customization.* The second high-level requirement the architecture must support is the capability to build customized solutions. This includes the ability to introduce new architecturally compliant tools into the solution in a "plug-and-play" manner, as well as the ability to customize the solution to the process used within the testing group. New tools are introduced by simply adding their plug-in code to the plug-ins directory for the workbench, or installing the tool if it is not a GUI-based tool. As long as the necessary integration components are available on the platform, the tool will be ready to run in the architecture.

Process customization is supported via the STAX service. The following scenario shows how this capability is used:

1. The tester creates a STAX workflow description, which defines a specific process that the tester wishes to automate.
2. The STAX description is stored as a resource on the workbench.
3. When the tester wishes to activate the process, he or she invokes it using the control-integration plug-in, which starts the STAX service on the description. The steps in the description are carried out. Workflow process invocation can also be automated, occurring when a particular event is detected.
4. The tester may review the outcome of the various steps in the process within the workbench GUI.

**Current status of the architecture.** We have developed the four integration components that comprise the architecture. The data integration component exists in prototype form, and we are currently working on enhancing and extending its capabilities. Many groups in IBM have deployed and are using the control integration component (STAF) and the GUI integration component (the WebSphere Studio Workbench) in production mode. We have defined in draft form the basic STCL standard artifacts (including test cases, test suites, hardware configurations, and related data) and we are preparing to circulate these definitions across IBM and to other interested parties. Similarly, we are constructing a draft version of the STCL standard events, along with an ITE plug-in for the workbench.

To guarantee that we are meeting the real requirements of IBM testers, we are doing pilot roll-outs of beta technology to three large IBM testing organizations. Feedback from these beta deployments will drive enhancements to the architecture and guarantee that solutions built using it meet the needs of the testing organizations.

## Conclusions

Because testing tools vary widely across the labs within IBM, our goal is to develop an architecture and supporting frameworks so that tool owners can easily integrate their tools with others that comply with the architecture. The ultimate goal is to provide an integrated testing environment. At a high level, there are three requirements that the architecture must satisfy:

1. Integrate the various STCL and other testing tools
2. Support easy installation and customization to meet site-specific processes
3. Support legacy tools

The paper offers two different visions of integration with the architecture. One addresses the design concerns that new tool developers should consider in order to be architecturally compliant. The other addresses the capabilities we are currently developing for making legacy tools compliant with the architecture. We demonstrate that the components we describe provide a basis for meeting the requirements defined earlier.

Development of the architecture is ongoing, with a prototype framework for the data integration portion completed. The control and GUI integration frameworks are deployed within IBM, but the standard definitions for extensible artifacts, events, and the ITE plug-in are still evolving. Our work has been greeted with considerable excitement by both the testing and tools development communities at IBM. It is our hope that the architecture will provide a flexible, sound basis for designing and developing test tool solutions that enhance the effectiveness and efficiency of the testing process across IBM.

## Acknowledgments

The authors gratefully acknowledge the work of numerous STCL members and other IBM employees who have served on the STCL Architecture Committee: Wayne Carrigan, Ian Craggs, Eric Crane, Eitan Farchi, Ian Griffiths, Howard Hess, Paul Kram, Hwaam Lee, Al Majko, Tom Messmore, Geoff Miller, Naomi Mitsumori, Simon Rushton, John Swanson, Shmuel Ur, and Andrew Wack. We would also like to thank Stan Sutton for his careful reading and excellent advice on the structure and content of the paper.



## Cited references

1. V. Gruhn and U. Wellen, "Integration of Heterogeneous Software Architectures—An Experience Report," *Software Architecture*, P. Donohoe, Editor, Kluwer Academic Publishers, Boston, MA (1999).
2. K. Bohrer, V. Johnson, A. Nilsson, and B. Rubin, "Business Process Components for Distributed Object Applications," *Communications of the ACM* **41**, No. 6, 43–48 (June 1998).

3. J. Amsden, "Levels of Integration," available at http://www.eclipse.org/articles/index.html.
4. *Reference Model for Frameworks of Software Engineering Environments*, Edition 3, National Institutes of Standards and Technology (NIST), Gaithersburg, MD (1993). ECMA is an international industry association for standardizing information and communication systems.
5. M. Chen and R. J. Norman, "A Framework for Integrated CASE," *IEEE Software* **9**, No. 2, 18–22 (March 1992).
6. P. Kruchten, "The 4 + 1 View Model of Software Architecture," *IEEE Software* **12**, No. 6, 42–50 (November 1995).
7. See http://www.fpml.org/.
8. J. Whitehead and M. Wiggins, "WEBDAV: IETF Standard for Collaborative Authoring on the Web," *IEEE Internet Computing* **2**, No. 5, 34–40 (September/October 1998).
9. C. Rankin, "Software Testing Automation Framework," *IBM Systems Journal* **41**, No. 1, 126–139 (2002, this issue).
10. See http://www.eclipse.org/.

*Accepted for publication October 5, 2001.*

**Clay Williams** *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (electronic mail: clayw@us.ibm.com).* Dr. Williams is a research staff member in the Center for Software Engineering at the IBM Thomas J. Watson Research Center. His research interests include model-based testing, software architecture, and software modeling languages. He is also interested in applications of computer science to medicine. Prior to joining IBM, he worked in the Clinical Information Services group affiliated with the Department of Medical Informatics at Columbia-Presbyterian Medical Center.

**Harm Sluiman** *IBM Application & Integration Middleware Division, IBM Canada, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 (electronic mail: sluiman@ca.ibm.com).* Mr. Sluiman is a Senior Technical Staff Member and development manager in the Application Development Technology Center at the Toronto Laboratory. His responsibilities cover relational database design tools, performance profiling, and system monitoring, as well as the metamodel of the application development life cycle, specifically testing. He has been an IBM employee since 1974 and worked in hardware customer service before joining the software development group in 1983. Since joining the lab he has worked on various types of software projects across all IBM platforms, primarily development tooling.

**David Pitcher** *IBM Application & Integration Middleware Division, IBM United Kingdom Laboratories, Hursley Park, Winchester, Hampshire, SO21 2JN, United Kingdom (electronic mail: David.Pitcher@uk.ibm.com).* Mr. Pitcher is an advisory software engineer in the Transaction Processing department at the IBM Hursley Laboratory working on testing strategy and model-based testing. He is currently technical lead for consolidated service test for z/OS and subsystems. Mr. Pitcher has worked on WebSphere, MQSeries™, TXSeries™, and most recently CICS™ testing technologies. He has a keen interest in formal methods, distributed systems, software design, and artificial intelligence.

**Marius Slavescu** *IBM Application & Integration Middleware Division, IBM Canada, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 (electronic mail: slavecu@ca.ibm.com).* Mr. Slavescu is a software engineer staff member in the Application Development Technology Center at the IBM Toronto Laboratory. His interests include network-based computing systems, software architecture, and automated testing. He is also interested in network security, Internet technologies, and operating systems architecture. Prior to joining IBM, he worked in the information technology department of The Canadian Depository for Securities Limited.

**Jim Spratley** *IBM Global Services, IBM UK Headquarters, Portsmouth, Hampshire, PO6 3AU, United Kingdom (electronic mail: jim_spratley@uk.ibm.com).* Mr. Spratley handles a suite of MVS- and Lotus Notes-based testing tools for the Testing Services group in IBM Global Services. He joined IBM in 1989 and has wide experience with the application development cycle, including development, technical support, and configuration management and testing.

**Mark Brodhun** *IBM iSeries Development, 3605 Highway 52 North, Rochester, Minnesota 55901 (electronic mail: brodhun@us.ibm.com).* Mr. Brodhun is a staff software engineer for the iSeries Integrated File Systems design and test team in Rochester. His most recent assignments included designing an improved test environment for the iSeries file system test and developing microcode debugging tools. He has a long-standing interest in the application of software tools to software development, specifically in the areas of end-user productivity and usability.

**John McLean** *IBM Canada, 8200 Warden Avenue, Markham, Ontario, Canada L6G 1C7 (electronic mail: jmclean@ca.ibm.com).* Mr. McLean is an advisory development analyst with the IBM Toronto Software Laboratory. In his 12-year career at IBM he has focused primarily on the testing of commercial software products, and he recently moved to development work for software testing tools. Prior to joining IBM, he held various development and test lead roles at EDS, Sears, and Air Canada.

**Charles Rankin** *IBM Server Group, 11401 Burnet Road, Austin, Texas 78758 (electronic mail: rankinc@us.ibm.com).* Mr. Rankin is an advisory software engineer in the IBM Austin Development Laboratory. He graduated with a B.S. degree in electrical engineering from the University of Florida in 1993, after which he joined IBM in Austin. He has worked extensively with IBM's PC-oriented operating systems and networking products. He was the system test lead for IBM's Directory and Security Server for OS/2 and IBM's OS/2 WARP Server for e-Business. He is currently the lead developer for STAF.

**Karen Rosengren** *IBM Corporate Division, 11400 Burnet Road, Austin, Texas 78758 (electronic mail: krosengr@us.ibm.com).* Ms. Rosengren is the IBM Software Test Chief Technologist reporting to the Corporate Director of Software Test. Her job entails finding and deploying the best possible practices in software testing to improve the software test community within IBM. She has been employed by IBM for 22 years and has an extensive background in software testing across multiple IBM products.