The ToolTalk Service

Preface

SunSoft™ Technical Reports are intended to help technical readers evaluate SunSoft products or technologies. In orientation and level of detail, they fill the gap between brochures and manuals. This report describes the ToolTalk™ service, which has been developed to facilitate message-based cooperation between independently developed application programs.

The ToolTalk service is a network-spanning inter-application message system. Cooperating application developers agree on a message protocol (the syntax and meaning of a set of messages), and use the ToolTalk service to deliver the messages.

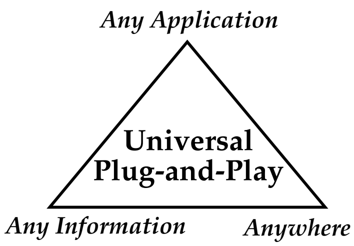
Applications create, send, and receive messages by calling ToolTalk functions from the ToolTalk application programming interface (API) library. The API gives the ToolTalk service the appearance of a simple library, hiding the details of its implementation; libraries included in applications interact with a collection of communicating processes running on machines throughout a local area network. This report:

- Provides an overview of the ToolTalk service.
- Introduces the *universal plug-and-play* application integration requirement.
- Describes the ToolTalk service communication facilities from a functional (not programming) point of view.
- Explains how applications interact with the ToolTalk service to send and receive messages.
- Discusses the standard sets that integrate ToolTalk compliant applications.

For purposes of simplification, this report takes some minor liberties with ToolTalk terminology. For the definitive description of the ToolTalk service, see the *ToolTalk Programmer's Guide*.

You should be generally familiar with workstations, networks, the UNIX® operating system, and interprocess communication. Programming examples in this report assume a reading knowledge of the C language.

Introduction



As computer users increasingly demand that independently developed applications work together, “interoperability” and “integration” are becoming important themes for software developers. These terms convey the idea of inter-application cooperation; of applications achieving more by working together than by working separately. The ToolTalk service, from SunSoft, a network-spanning inter-application message system, has been designed to promote cooperation among independently developed applications.

Use the ToolTalk service when you want *universal plug-and-play* capability. The term *universal plug-and-play* means that any tool can be replaced by any other tool that follows the same protocol. That is, any tool that follows a given ToolTalk protocol can be placed (plugged) into your computing environment and can perform (play) those functions indicated by the protocol. Tools can be mixed and matched, without modification and without having any specific built-in knowledge of one another. Universal plug-and-play is built on the concept of describing applications by the functions the addressee needs to perform, rather than calling a specific type of application.

The ToolTalk service is completely distributed and network aware. Location issues are entirely transparent to a user. While it provides easy global access, ToolTalk hides all networking details. The ToolTalk service can function as a partner to Open Windows or can operate independently of any window system. In addition to this, ToolTalk can be retrofitted to any existing application.

The ToolTalk service is a standard feature of the Solaris 1.0 and 2.0 operating systems software platforms. All Solaris users have ToolTalk installed and available on their systems.

ToolTalk messaging is an API of the DOE Application Framework. If you adopt ToolTalk message compliant applications today, you can be assured of having made an excellent investment. You will be able to continue to use your existing applications in the DOE Application Framework as you enhance your applications to take advantage of its distributed object facilities.

Universal Plug-and-Play: The Application Integration Requirement

Today, computer users have access to a wide variety of applications that are either purchased from software suppliers or developed in-house. Not only do users want to use applications they are already familiar with, they want their tools of choice to be integrated with other applications automatically. Typically, a company limits the choices available to its workers by standardizing on one or more applications of a particular type. Unfortunately, this results in little integration among applications. This is not the ideal solution. Without the best, highly integrated tools, an organization does not have an edge in the global market.

Users want any and all applications to integrate automatically and consistently. They need automated access to all data types including numerical, text, graphical, audio, video, and control information. And, they should not be required to know where the information is stored. This is the *universal plug-and-play* requirement of applications, a theme depicted in Figure 1.

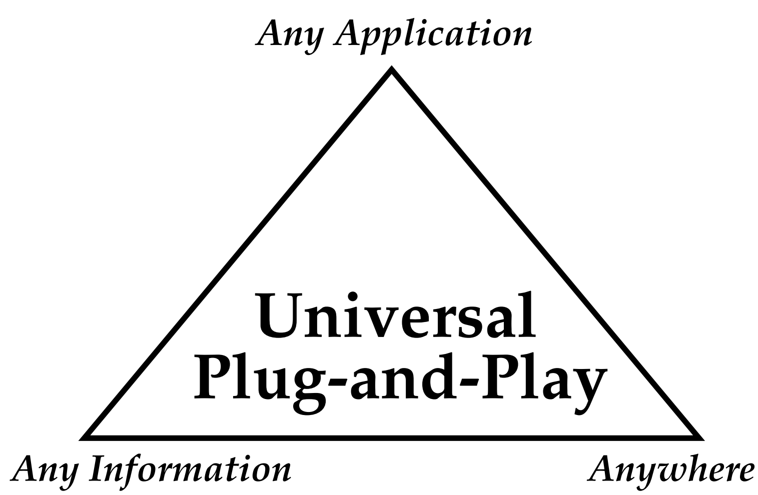


Figure 1 Universal Plug-and-Play: The Application Integration Requirement

ToolTalk meets the universal plug-and-play integration requirement. It:

- Supports message passing between any applications
- Accesses any information types
- Integrates workgroups and provides global access while hiding the networking details
- Makes messaging entirely transparent to users
- Provides high performance and throughput

Applications that use the ToolTalk service cooperate by sending and receiving messages. The messages passed by cooperating applications form a small, well-defined message protocol that maximizes both application integration and autonomy. Many applications behave as though they are just a single application.

ToolTalk supports procedural and object-oriented messaging. It leverages the functionality of other applications and fits easily into the mix of procedural applications that dominate current usage.

If message protocols are observed, cooperating applications can be modified — even rewritten — without affecting one another.

Some advantages of the inter-application communication provided by the ToolTalk service are:

- Applications can do the tasks they do best and ask other applications to perform tasks for which they have little or no functionality.
- Inter-application communication is a small investment with the ToolTalk service. Existing applications are easily modified to send, receive, and use shared information.
- No knowledge about the application receiving the information is needed — including its location on the network. The ToolTalk service determines who receives the information by looking for applications who asked for the type of information being sent.
- Role addressing enables addressing to be generic. It describes the functions the addressee needs to perform, such as editing, instead of calling a specific named application. The appropriate application is then addressed.

Developers of interoperating applications establish a message protocol and use the ToolTalk service to deliver the messages. The ToolTalk service supports many protocol styles. A sender can address a ToolTalk message to a particular process or to any process. Message senders are unconcerned with the locations of processes in a local area network; the ToolTalk service finds receiving processess. The messages in a protocol typically convey control and environment change information. For example, file creation/deletion, and a request for display.

Use the ToolTalk service when your application requires *control integration*. The term control integration indicates a group of tools working together toward a common end without direct user intervention. The ToolTalk service enables control integration through its easy and flexible facilities for issuing arbitrary requests, either to specific tool instances or to anonymous service providers.

Applications interact with the ToolTalk service by calling functions defined in its application programming interface (API). The API gives the ToolTalk service the appearance of a simple library, hiding the facts of its implementation as both a library and a collection of communicating processes running on computers throughout a local area network.

Why the ToolTalk Service?

The ToolTalk service has been developed to permit processes to communicate in a variety of ways with minimal knowledge of each other and no knowledge of the local area network. Acting as a message switch, it is the ToolTalk service, rather than message senders and receivers, that tracks process identities and locations (see Figure 2). Receivers register *patterns* that describe the kinds of messages they want to receive. Although a message sender can name the receiver of a message, it usually does not; instead, it lets the ToolTalk service deliver copies to interested receiver.

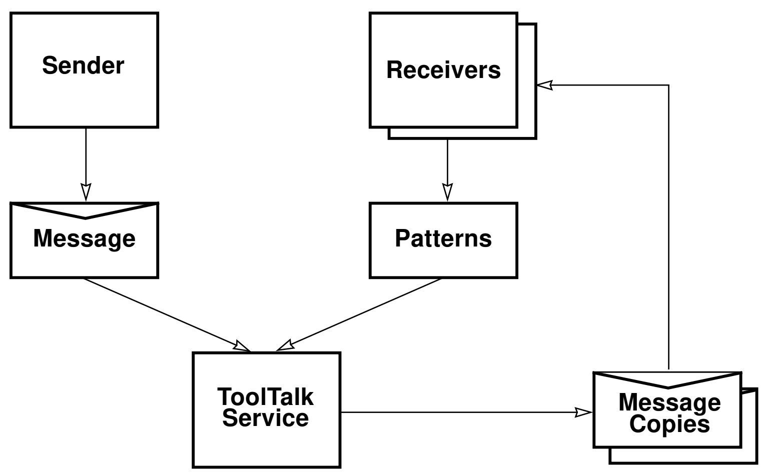


Figure 2 Message Switching

For example, one process wants to inform all interested processes running in behalf of a user that an event has occurred. Because network window systems allow users to run processes on multiple machines, it is difficult for the sending process to know which processes should be notified and where they are running. Using the ToolTalk service, the sending process simply sends a message *scoped* to (a way of addressing) the *session* (group of applications active in a particular ToolTalk instance). The ToolTalk service delivers copies of the message to all processes in the session that have registered a pattern that matches the message.

Capabilities of the ToolTalk Service

Some highlights of the capabilities of the ToolTalk service:

- Message addressing is very flexible. A message can be addressed to a particular process or to any process (which may include a group of processes). A message can also be addressed to a process that can perform a particular operation. It does this by describing the functions that need to be performed, rather than calling a specific type of application. This is *role addressing*. For example, instead of calling a particular spreadsheet application, the process that perform data manipulation is addressed.
- Processes can send notice messages that announce events and request messages that ask for operations to be perfomed. Multiple processes may receive copies of either kind of message for observation, but only one receiver can accept and execute a request.
- An application developer can register aa applications's interest in receiving messages when no process is running the application. A developer can further direct the ToolTalk service to start a process when a message becomes available, or to save the message until a process is started by another resource.

To implement these facilities, the ToolTalk service interacts with four main entities:

1. Applications (sometimes called tools)
2. Processes
3. Messages

Applications, Processes, and Sessions

One execution of an application that calls ToolTalk API functions is a *process*. A ToolTalk process is a UNIX process about which the ToolTalk service has some extra information. A UNIX process can actually contain multiple ToolTalk processes. Unless otherwise stated, the word *process* in this paper indicates a ToolTalk process.

Processes are grouped into sessions; a session usually represents the processes displaying on one user's desktop. The processes in a session may be running on different computers in the network, using a distributed window system to coordinate their displays on the user's desktop. A ToolTalk message can be multi-cast to the processes in a session, giving a user's processes a convenient way to inform one another of events.

ToolTalk messages are scoped to limit their potential distribution. There are four ways to scope a message:

1. by session
2. by file
3. by the union of file and session
4. by the intersection of file and session

The processes displaying on one user's desktop are eligible to join and, therefore, receive messages scoped to the user's session. A session-scoped message is delivered to only the processes in the session named in the message.

As a process opens and closes files, it can inform the ToolTalk service that it is interested in messages scoped to its open files, and not interested in messages scoped to files it has closed. A file-scoped message is delivered to only the processes that have expressed interest in the file named in the message.

Within a message's scope, the ToolTalk service determines the actual receivers by matching *attributes*. Each message consists of attributes such as:

- Address (procedure, process)
- Class (request or notice)
- Operation (action to be performed, or event that has occurred)
- Arguments (input values for operation, output results)
- State (created, sent, handled)

Sending processes supply values for message attributes. Receiving processes specify, as patterns, the attribute values of messages they wish to receive. The ToolTalk service compares message attributes to receiver patterns, and delivers copies of a message to processes that have registered matching patterns.

Messages

Processes use the ToolTalk service to send messages to one another. The set of messages that a group of processes exchange, together with the definitions of when messages are sent and how they are processed when received, constitute a message protocol.

Notices and requests

Processes can send two classes of ToolTalk messages, notices and requests. A notice is informational — a way for a process to announce an event. Processes that receive a notice absorb the message without returning results to the sender. A request is a call for an action. The results of the action are recorded in the message and returned to the sender as a reply.

Handlers and observers

For each pattern describing a message it wants to receive, a process also declares whether it can *handle* the message or only wants to *observe* it. Both requests and notices can be handled or observed. Although many processes can observe a message, only one process can handle it. Thus, a requested operation is performed only once. If the ToolTalk service cannot find a handler for a request message, it returns the message to its sender with a state of failed. Because notices do not return results, an unhandled notice is not an error.

Routing

A notice takes a one-way trip, as shown in Figure 3. The sender creates a message, fills in attribute values, and sends the message. The ToolTalk service matches message and pattern attribute values, and gives a copy of the message to one matching handler and to all matching observers. File-scoped messages are automatically transferred across session boundaries to processes that have declared interest in the file.

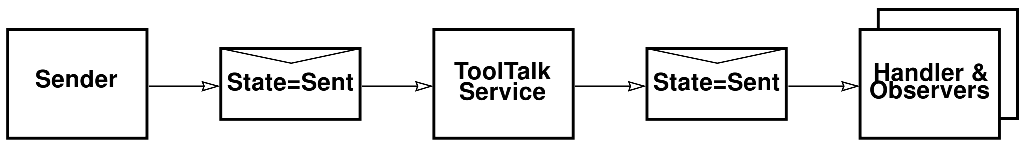


Figure 3 Notice Routing

A request, as shown in Figure 4, takes a round-trip from sender to matching handler and back to the sender. Copies of the message take one-way side trips to matching observers. The ToolTalk service delivers a request to only one matching handler. The handler adds results to the message and returns it to the sender. Other processes can observe a request before it is handled, after it has been handled, or at both times; observers absorb a request without returning it back the sender.

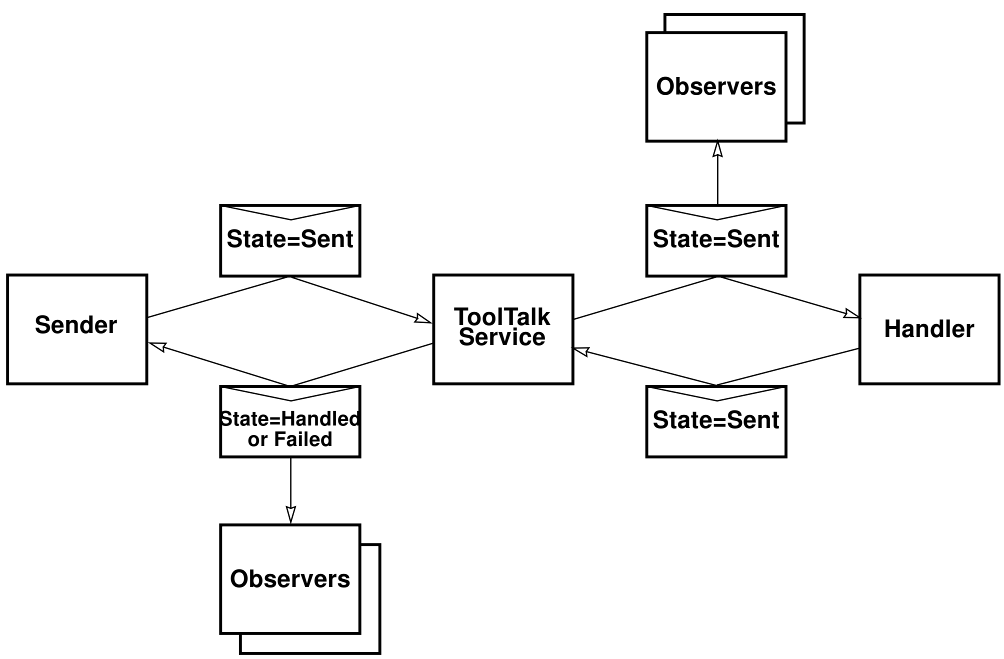


Figure 4 Request Routing

Addressing

Although messages are ultimately delivered to processes, they can be addressed to processes or procedures. Each ToolTalk process that receives messages has a process identifier (*procid*) that is unique within a session. A sender that knows a receiver's procid can send a process message which specifies the receiver.

However, usually a sender does not care which process executes an operation (request message) or which processes learn of an event (notice message). The message is effectively addressed "to whom it may concern." The sender specifies an operation and arguments; the ToolTalk service finds, by pattern matching, one handler and all observers.

What the ToolTalk Service Does

This section first describes the fundamental elements of the ToolTalk message service: messages and processes. It then describes how messages are delivered to processes.

As shown in Figure 5, a process called TTsession is the switch that routes messages from sender to receiver processes. Using ToolTalk API functions, application processes send messages to TTsession and receive messages from it. Each session has its own instance of the TTsession process: a file-scoped message, which may cross the sender's session boundary, is forwarded by the sender's TTsession to other TTsessions which deliver it to interested processes in their sessions.

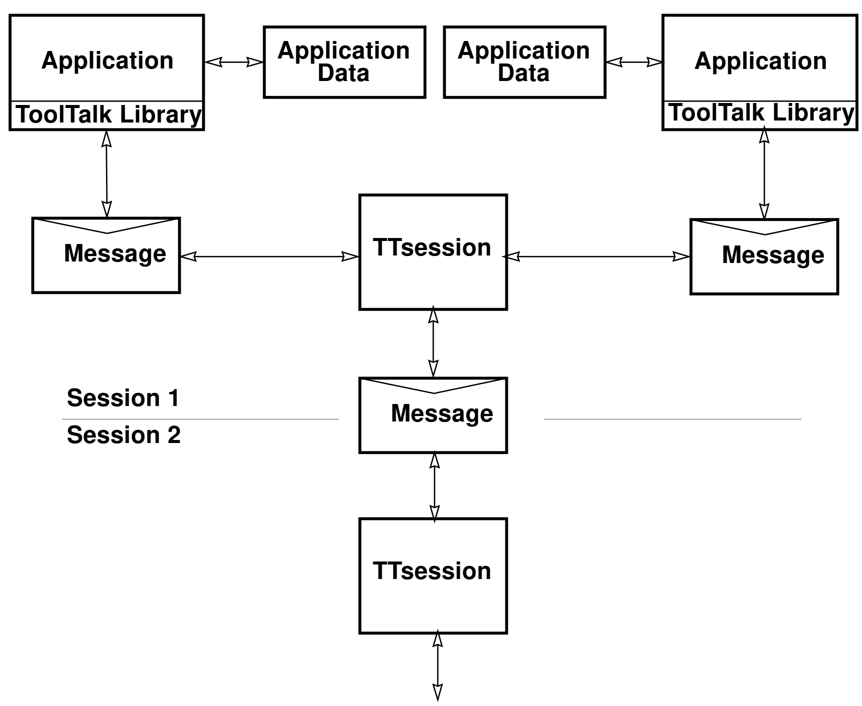


Figure 5 ToolTalk Message Flow

Messages

ToolTalk messages have a single format which is summarized in Table 1. The fields in a message are called attributes, and the contents of the fields are called values.

Table 1 Summary

Attribute	**Values**	**Description**
Address	Procedure or Process	Entity to which message is addressed
Class	Request or Notice	Reply is or is not required
State	Created, Sent, Handled, Failed, Queued, Started	Message’s delivery status
Status	(code)	Explanation of failed message
Operation	(op)	Operation (request) or event (notice)
Opnum	(opnum)	Disambiguates multiply defined ops
Arguments	(args, results)	Operation arguments and results
Sender	(procid, ptype)	Sender’s identity
Handler	(procid, ptype)	Receiver’s identity
Scope	Session, File, Both, File-in-Session	Potential receivers
Session	(sessionid)	Sender’s session
File	(pathname)	File associated with object
Disposition	Queue and/or Start	Disposition if no receiver is running

Although all messages have the same format, four principal kinds of message can be distinguished by their address values:

1. **Procedure**: The operation attribute specifies either an operation to be executed by a process unknown to the sender (that is, a remote procedure call), or an event to be relayed to processes unknown to the sender.
2. **Process**: Similar to a procedure message except that the sender specifies (in the handler procid attribute) the identity of the receiving process.

A message's address attribute determines what attribute values a sender supplies, and how the ToolTalk service delivers the message.

Note – The sender supplies the identity of the receiving process for only one address type (process). For the other address types, TTsession determines the identities of the receivers.

Message senders, TTsession, and message handlers can supply values for message attributes (see Figure 6). The sender supplies addressing, operation, and argument values. TTsession adds the sender's procid and values from matching *signatures* (descriptions of procedures) and updates a message's state value as delivery progresses. A request handler updates a message's arguments with operation results. The updated message is returned as a reply to the sender.

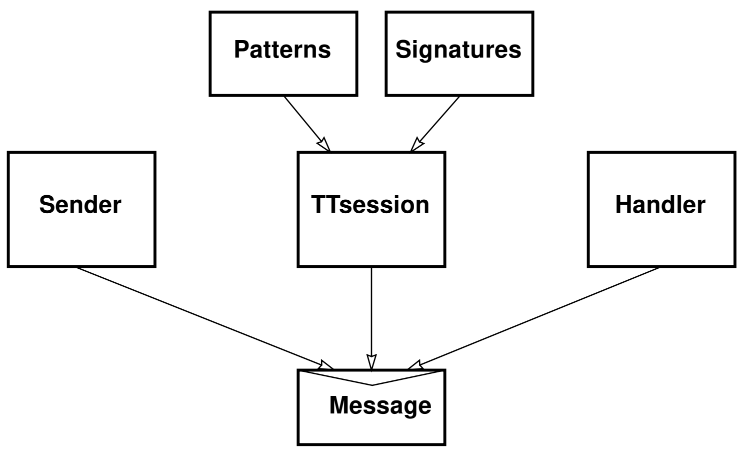


Figure 6 Attribute Value Sources

Processes

An instance of a running application that uses the ToolTalk service to send or receive messages is called a process. A ToolTalk process is a UNIX process for which the ToolTalk service has extra information. A process provides some of this information by calling ToolTalk functions; other information comes from a data structure called a process type (*ptype*).

ptypes

An application developer can direct the ToolTalk service to consider the application as a potential message receiver when no process is running the application. To do this, the developer describes the messages of interest in a ptype. The syntax of a ptype definition resembles the syntax of a C++ class definition; Figure 7 shows a simple example.

```
ptype CellCalc {
   start "/usr/bin/cellcalc";

   observe:
      file saved() => queue;
   handle:
      session show_cell(in int row, in int col)
      => start opnum=10;
      session show_cell(in string name)
      => start opnum=20;
};
```

Figure 7 Simple ptype

A ptype begins with an identifier known as a process type identifier (*ptid*); `CellCalc` in Figure 7 is a ptid. The optional *start string* that follows the ptid is a command that a TTsession process executes to start a process running the application; `/usr/bin/cellcalc` in Figure 7 is a start string.

Following the start string are signatures describing the procedure and process messages that the application wants to receive. Messages to be observed are described separately from messages to be handled. A signature is divided into two parts by an arrow (=>). The first part specifies matching attribute values; for example, the observe signature in Figure 7 specifies

> *match all messages whose scope is file, whose operation is saved, and whose arguments are anything*

The handle signatures in Figure 7 specify that only session-scoped messages are to be considered. The more attribute values specified in a signature, the fewer messages the signature will match.

The second part of a signature specifies receiver values that the ToolTalk service will copy into messages that match the first part of the signature. A ptype signature can contain values for *disposition* and *opnum*. The ToolTalk service uses the disposition value (start, queue, or discard) to determine what to do with a message that matches the signature when no process is running the application. The opnum value is provided as a convenience to message receivers. Whentwo signatures have the same operation name but different arguments, giving them different opnums makes incoming messages easy to identify. In Figure 7, for example, a receiver needs only to look at the message's opnum attribute to determine what arguments are in a `show_cell` message.

Patterns

A pattern is a set of attributes and values that expresses the matching messages a process wants to receive. A process has two sets of patterns: one pattern defines the messages it wants to observe; the other pattern defines the messages it wants to handle. Figure 8 shows three simple patterns.

```
Observe
   (Observer_Ptype: "BogusCalc";
    Class: notification;
    Op: "saved";
    Scope: file;
    File: "/etc/boatspex/boatdata"
       "/home/skippy/billc/.boatsrc";)
Handle
   (Handler_Ptype: "BogusCalc";
    Class: request;
    Op: "show_cell";
    Scope: session;
    Session: 999;)
   (Handler_Ptype: "BogusCalc";
    Class: request;
    Op: "make_chart";
    Scope: session;
    Session: 999;)
```

Figure 8 Simple Patterns

A process can register patterns *statically*, *dynamically*, or both. The signatures in a ptype are equivalent to statically declared patterns; after a process with a ptype begins to run, it declares its ptype by calling a ToolTalk function. In response, the ToolTalk service creates patterns from the corresponding ptype's signatures. If the process needs additional patterns, it can create them dynamically by calling ToolTalk functions. A process that has no ptype registers patterns dynamically by calling these same functions.

Dynamically registered patterns have two capabilities that statically registered patterns (that is, those defined in signatures) do not:

1. A dynamically registered pattern can be unregistered (and registered again), giving a process the ability to receive a class of messages temporarily.

2. A *callback* function can be associated with a dynamically registered pattern. A callback function is an application function that is invoked automatically when a message matching the pattern is received. Incoming messages that do not have callback functions can be more meaningful by examining their attribute values, particularly the opnum attribute.

The ToolTalk service automatically updates a process's patterns as it calls ToolTalk functions that *join* and *quit* sessions and files.

- Joining a session makes a process eligible to receive messages whose scope is that session.

- Joining a file makes a process eligible to receive messages whose scope is that file.

For example, if the process whose patterns are shown in Figure 8 joins a file called `/etc/project/specs`, the ToolTalk service will add this file name to the process's observe pattern. (The handle patterns, whose scope is session, will not be changed.) A ToolTalk process can join only one session; however, a UNIX process can have multiple ToolTalk processes.

Sending and receiving

To send a message, a process calls ToolTalk functions to *allocate* an empty message, fill in attribute values, and send the message. Allocating a message returns a unique identifying handle for the message. A reply to a request, which is received as any other message, can be identified by its handle which is the same as the corresponding request's handle. (It is also possible to associate a callback function with a request which will be called automatically when the reply comes in. The “Application Programming Interface” section provides details.)

When the ToolTalk service has a message available for a process, it activates a UNIX file descriptor created by the process. A process can check the file descriptor directly or have its window system toolkit treat activation of the file descriptor as a window system event. After it has been notified that a message is available, a receiving process calls a ToolTalk function that returns the message’s handle. Other ToolTalk functions can then be called to return message attribute values.

Message Delivery

The ToolTalk service delivers messages according to their address and scope values. These two attributes can take numerous combinations of values; this section covers the main combinations:

- session- and file-scoped procedure messages
- process messages
- request replies

Note – Message delivery as described here is functionally accurate but ignores some complicating optimizations.

Although each combination of address and scope values is delivered differently, three mechanisms are employed in many delivery scenarios:

1. **Signatures**. An application can express interest in particular messages by specifying *signatures*. Signatures are patterns that match the names and arguments of messages of interest. When a TTsession receives a message from a sender, it first looks for matching handle and observe signatures and creates a copy of the message for each match. The TTsession delivers the matching copies to processes running the corresponding applications. Leftover message copies (representing matches with applications that no process is running) are handled according to the disposition value in the corresponding signature.

2. **Session records**. A file-scoped message is delivered to all observers that are joined to the file and interested in the message—no matter in which session the processes are running. The search for a message handler does not stop in

the sender's session; if no handler is found there, the other sessions are searched. To effect inter-session delivery, the TTsessions maintain a special spec for each file that has been joined; at any time the spec lists the identities of the sessions that have at least one process joined to the file. By searching this spec, a message sender's TTsession can forward a message to other TTsessions that might have a handler for the message.

3. **Message queues**. File-scoped messages which cannot be delivered when they are sent, and which have a disposition of queue, are held in queues associated with file specs. When a process joins a file, the process's TTsession dequeues the messages queued on the file spec and matches them as if they had just been sent. The TTsession delivers messages that match and requeues those messages that do not.

Procedure Messages

A session-scoped procedure message is delivered by a TTsession as follows:

1. Search the ptypes for a matching handle signature based on scope, class, operation, and arguments.
   - If a match is found, copy the handler ptype and disposition from the signature to the message. There can be only one match because each combination of operation and arguments is unique across ptypes.
   - If a match is not found and the message is a notice, discard it.
   - If a match is not found and the message is a request, return the message to the sender with a state of failed.

2. For the handler copy, search the patterns for all matches.
   - Rank the matches by attributes matched. (Wildcard matches—those that allow any value for an attribute—rank lower.)
   - Notify the best-matching process that a message is available. If more than one process have the same number of matching attributes, one of these processes is arbitrarily choosen to receive the message.
   - If the message is a request, save a "reply pending" copy.

3. If a signature for a handler was matched but no handler pattern matches the message, follow the message disposition copied from the signature. However, if the message is a request and the deposition is discard, return the message to the sender with a state of failed.

4. Search the ptypes for matching observe signatures. For each match found, copy the observer ptype and disposition from the signature to the message.
5. For each observer copy, search the patterns for a match:
   - If a match is found, notify the corresponding process that a message is available.
   - If no match is found, follow the message disposition.

A file-scoped procedure message is delivered similarly to the way session-scoped messages are delivered, with two additions:

1. The sender's TTsession marks a copy of the message "for observers only" and sends it to each TTsession that is joined to the file.
2. If no local handler is found, the sender's TTsession forwards the message marked "handler only" to one other TTsession that is joined to the file.
   - If the receiving TTsession does not have a matching handler, it returns the message to the sending TTsession as failed.
   - The sender's TTsession then forwards the message to another TTsession that is joined to the file.
3. If there is no local handler and no TTsession joined to the file accepts the message, the sender's TTsession follows the message disposition copied from the signature. However, if the message is a request and the deposition is discard, it returns the message to the sender with a state of failed.

Process messages

A process message (one in which the sender specifies the receiver's procid) is delivered similarly to the way a procedure message is delivered; however, the signatures and patterns are not searched for matching handlers.

Reply messages

A TTsession process generates a reply message whenever it changes the state attribute of a request. A TTsession most commonly changes a request's state when a handler calls a ToolTalk function that signifies "this message has been handled" or "this message has been failed". In these cases, the TTsession updates message arguments with results provided by the handler and changes the message's state to handled or failed. A copy of the reply is sent directly to the sender. The TTsession also matches the reply message's attribute values against the observe patterns of running processes and sends a copy of the reply to the processes with matching patterns.

The ToolTalk service also changes a request's state when it queues a message for later delivery and when it starts a process running the request handler. Thus, a request sender may receive multiple replies to a single request message because the replies have the same handle as the original request. For example, if a message disposition is queue and no process is running the application that handles the request:

1. The sender will receive a reply to the message when the ToolTalk service queues the message.
2. The sender will receive a second reply to the message when a process running the handler application is started.
3. The sender will receive a thirdreply to the message when the handler performs the request and replies to the message.

Senders not interested in the intermediate states of a request can ignore replies whose state is not handled or failed.

Application Programming Interface (API)

Applications interact with the ToolTalk service by calling functions defined in the ToolTalk API. The following directive includes the function interfaces in a C or C++ program:

```
#include <desktop/tt_c.h>
```

This section describes the major functions used for process management; patterns; and sending and receiving messages and replies. The examples shown in this section are untested fragments that are intended only to give a concept of ToolTalk programming.

Process Management

The process management functions listed in Table 2 are typically called by a ToolTalk application that is either initializing itself or preparing to terminate.

Table 2 Process Management Functions

Function	Description
tt_open	Get procid
tt_close	Release TTsession process resources
tt_fd	Get file descriptor for incoming message signal
tt_declare_ptype	Build patterns from ptype

`tt_open`
The first ToolTalk function a process calls. This function makes the process known to its TTsession and returns the process's procid.

`tt_close`
The function a process calls before terminating. This function frees the TTsession resources allocated to the process.

`tt_fd`
The function a process calls to create a UNIX file descriptor. The ToolTalk service signals that a message is available by activating this descriptor. The "Receiving a Message" section describes how a process can detect whether the file descriptor is active.

`tt_declare_ptype`
: The function a process calls to declare a ptype. Declaring a ptype instructs the ToolTalk service to build patterns for the process.

Figure 9 shows these functions in typical use.

```
char   *myprocid = tt_open();
int    ttfd;

/* check open */
if (tt_pointer_error(procid) != TT_OK) {
   return 0;
}

/* get file descriptor for incoming messages */
ttfd = tt_fd();

/* build patterns from ptype */
if (tt_ptype_declare("Demo") != TT_OK) {
   fprintf(stderr, "Demo ptype not installed. \n");
   return 0;
}
/* send and receive messages ... */

/* wind it up */
tt_close();
exit(0);
```

Figure 9 Typical Use of Maintenance Functions

Registering Patterns

To receive messages, a process must register at least one pattern with the ToolTalk service. A process can indirectly register patterns by declaring a ptype, or it can directly create and register patterns using the functions shown in Table 3.

Table 3 Pattern Functions

Function	Description
tt_pattern_create	Create empty pattern
tt_pattern_destroy	Reclaim pattern's storage
tt_pattern_register	Activate pattern
tt_pattern_unregister	Deactivate pattern
tt_pattern_category_set	Observe or handle matching message
tt_pattern_XXX_add	Add value for *XXX* attribute to pattern
tt_session_join	Add sessionid to session-scoped patterns
tt_session_quit	Remove sessionid from patterns
tt_file_join	Add file name to file-scoped patterns
tt_file_quit	Remove file name from patterns
tt_pattern_callback_add	Specify function for matching messages

To register a new pattern, a process creates an empty pattern, adds attribute values to it, and then registers the pattern. Patterns can also be unregistered (and re-registered), and deleted. Figure 10 shows an example of creating and registering a pattern that will match any session-scoped message whose operation attribute is `Jibe`; the registering process wants to handle matching messages.

```
Tt_patternpat;

pat = tt_pattern_create();
tt_pattern_op_add(pat, "Jibe");
tt_pattern_scope_add(pat, TT_SESSION);
tt_pattern_category_set(pat, TT_HANDLE);

if (tt_pattern_register(pat) != TT_OK) {
   return 0;
}
```

Figure 10 Creating and Registering a Pattern

Joining and quitting sessions and files alters a process's patterns.

- Joining a session adds the sessionid to all patterns whose scope is session, file-in-session, or both.
- Quitting a session removes the sessionid from all patterns.
- Joining a file adds the file name to patterns whose scope is file or file-in-session.
- Quitting a file removes the file name from all patterns.

When a ToolTalk application can receive more than one kind of message, it typically routes each kind of message to a different function. An incoming message can be automatically routed by attaching a callback function to a pattern. When a message matching such a pattern is received, the ToolTalk service calls the callback function, passing the message as an argument. For example, the code shown in Figure 11 directs the ToolTalk service to invoke a function named `jibe_callback` for each message that matches the pattern `pat`.

```
Tt_callback_actionjibe_callback;

tt_pattern_callback_add(pat, jibe_callback);
```

Figure 11 Establishing a Callback

The “Receiving a Message” section describes message receiving and callback functions in more detail.

Sending a Message

To send a message, a process follows a sequence that is similar to registering a pattern: it creates an empty message, gives values to message attributes, then sends the message. The major ToolTalk functions for sending messages are listed in Table 4.

Table 4 Sending Functions

Function	Description
tt_pnotice_create	Create a procedure notice
tt_prequest_create	Create a procedure request
tt_message_XXX_set	Set value of *XXX* attribute
tt_message_arg_val_set	Set argument value
tt_message_destroy	Release message’s memory
tt_message_callback_add	Specify function called for reply
tt_message_create_super	Create copy of message for supertype.

Figure 12 shows a process sending a simple procedure notice.

```
Tt_messagemsg_out;

msg_out = tt_pnotice_create(TT_SESSION, "tacked")
tt_message_arg_add(msg_out, TT_IN, "integer", NULL);
tt_message_arg_ival_set(msg_out, 0, (int)new_tack());
tt_message_send(msg_out);
tt_message_destroy(msg_out);
```

Figure 12 Sending a Simple Procedure Notice

Note – The process in this example destroys the message after sending it because the message is a notice and notices do not cause replies. A reply overwrites the memory occupied by the associated message; a process must not destroy a message that is a request until the matching reply has been received.

Receiving a Message

Table 5 lists the most important functions for receiving and examining ToolTalk messages.

Table 5 Receiving Functions

Function	**Description**
tt_message_receive	Get next message
tt_message_XXX	Get value of *XXX* attribute
tt_message_arg_mode	Get argument mode (in, out, inout)
tt_message_arg_type	Get argument type
tt_message_arg_val	Get argument value

The `tt_message_receive()` function returns a handle to the next message, or `NULL` if there is no message. There are three ways to use this function to receive messages:

1. Poll for messages by issuing `tt_message_receive()` in a loop until the function returns a handle.

2. Use the `select()` system call to learn when a message is available, then issue `tt_message_receive()` to retrieve it.

3. Direct a window system toolkit to call an application function when the ToolTalk file descriptor becomes active, and issue `tt_message_receive()` in that function.

Figure 13 shows a function that receives a message, processes it according to the message's opnum, and then destroys the message.

```
void next_msg()
{
   Tt_messagemsg_in;
   int bad_opnum;

   /* don't collide with TT status values */
   bad_opnum = TT_ERR_LAST + 1;
   msg_in = tt_message_receive();
   if (msg_in == NULL) return;
   switch (tt_message_opnum(msg_in)) {
      case EDIT_OPEN:
      /* process message*/
      break;
      default:
      /* don't recognize opnum */
      tt_message_status_set(msg_in, bad_opnum);
      tt_message_fail(msg_in);
      break;
}
tt_message_destroy(msg_in);
return;
}
```

Figure 13 Receiving a Message

If an incoming message has an associated callback function, calling `tt_message_receive()` has the effect of calling the callback function. This technique can be used to automatically route replies and messages that match dynamically registered patterns. Messages that match statically registered patterns (that is, those built from ptype or otype signatures) can be routed easily if their signatures specify different opnums; to specify different opnums, use the opnum in a `switch` statement, as illustrated in Figure 13.

A process receives a request reply the same way as any other message receives a reply. If a reply does not have a callback function, the reply can be identified by comparing the handle returned by `tt_message_receive()` with the handle returned by the function that created the message.

Sending a Reply

A process that handles a request updates the message's out and inout arguments with results, then returns the updated message. Functions used by replying processes are listed in Table 6.

Table 6 Replying Functions

Function	**Description**
tt_message_arg_val_set	Set value of message argument
tt_message_reply	Operation succeeded
tt_message_failed	Operation failed
tt_message_reject	Try another receiver
tt_message_status_set	Set status attribute explaining failure

`tt_message_reply()`
This function returns the message with its state set to *handled*.

`tt_message_failed()`
This function returns the message with its state set to *failed*.

`tt_message_status_set()`
This function indicates the reason for the failed message.

`tt_message_reject`
This function directs ToolTalk to attempt delivery to a different handler.

Standard Message Sets

Applications usually are not developed concurrently. And, typically they are developed by multiple groups. Users often want to substitute their favorite applications, so they want the universal plug-and-play across all applications. ToolTalk's standard message sets provide integration benefits to users of compliant applications.

By having messages organized into discrete sets, you get a pre-arranged interface between applications with common interests. These sets are specific to an application domain; for example, accounting or computer aided software engineering. Figure 14 represents the relationships among the message sets.

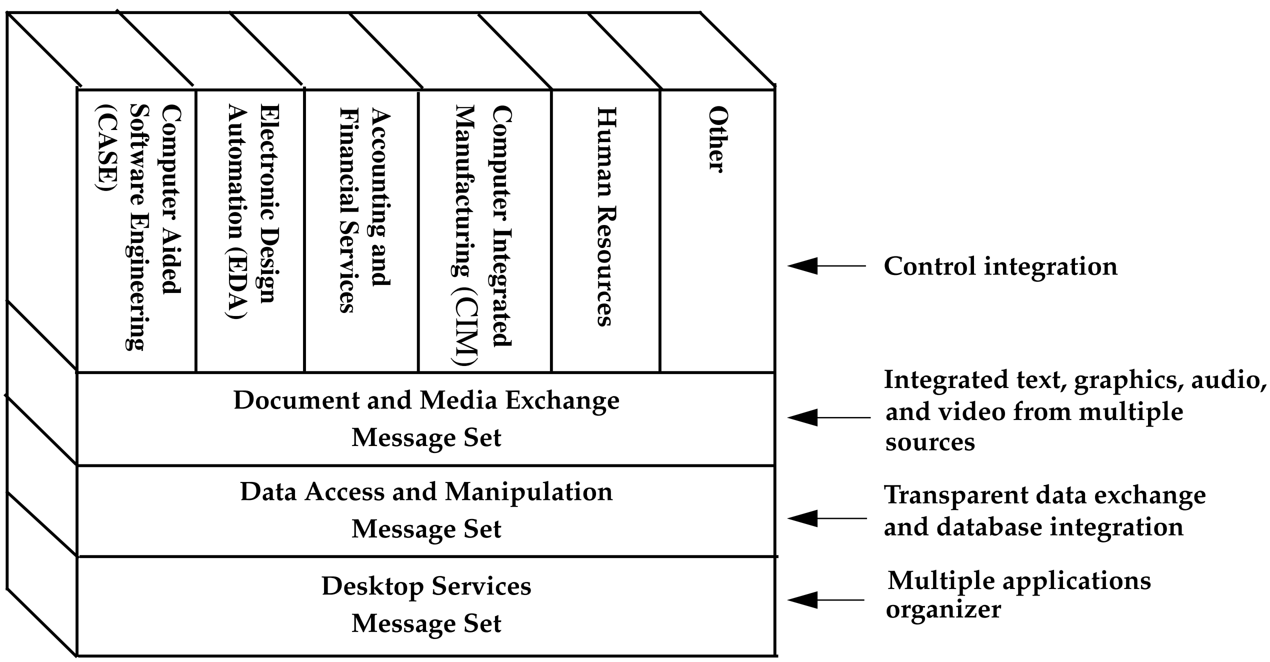


Figure 14 Standard ToolTalk Message Sets and Integration Benefit Examples

As the result of extensive work to define message standards with leading software suppliers and end users, SunSoft provides these well-defined, highly functional, standard message sets to anyone. They are and open specifications in the public domain without royalty or license fees.

ToolTalk compliant applications realize many integration benefits from the various message sets, which are described below.

Desktop Services Message Set

To achieve basic desktop integration, applications need to support a basic set of messages to enable inter-application control. With the Desktop Services Message Set, an application can manage the state of files, windowing, and processing for itself or any other application. A powerful messaging protocol that benefits both developers and users of desktop applications, the Desktop Services Message Set allows applications to interact easily with other desktop applications.

For integrated control of applications to take place, certain basic features are needed to launch, halt, control display appearance, and pass information regarding input and output data. All applications need to have these facilities so that other applications in the toolset can inter-change basic control information. This kind of functionality enables the development of smart desktops and integrated smart toolsets. Groups of applications can now call upon one another to perform tasks and to interact as one solution environment for the end user.

Data Access and Manipulation Message Set

Users who access databases and manipulate information in spreadsheets and decision support applications need transparent integration. The Data Access and Manipulation Message Set makes data integration seamless and transparent to users.

Document and Media Exchange Message Set

No single vendor provides a completely integrated solution that meets the complex needs of today's market. The ToolTalk Document and Media Exchange Message Set allows any application to incorporate text, graphics, audio, and video generated by another application. Virtually all applications can benefit from close integration of documentation and multimedia.

Using ToolTalk Document and Media Exchange Message Set, multimedia applications can communicate with one another in a transparent manner, both locally and over networks, regardless of data formats, compression technology, and other technical issues all of which have previously confined the use of this technology.

CASE Interoperability Message Sets

To achieve control flow integration in an integrated CASE environment, applications need to support messages that enable inter-application control. The CASE Interoperability Message Sets are the common message sets that provide this functionality for CASE applications. A powerful messaging definition that benefits both developers and users of CASE applications, the CASE interoperability Message Sets allow applications to interact easily with other CASE applications. The CASE Interoperability Message Sets define a standard set of CASE functionality available through control integration services. Each message set describes an area of functionality in the CASE environment. The message sets are:

- Analysis and Design
- Browsing
- Building
- Debugging
- Editing
- Version Management

Adopting Message Sets

It is easy to adopt ToolTalk messages. With help from a series of technical white papers, an experienced programmer can learn how to incorporate them into new or existing applications.

If you are a software vendor, you are not obligated to adopt all message sets. You can use some or all, depending on the functional integration needs of your customers. Your applications can then interoperate with any other application with the same message set(s).

Listing of Standard Messages Sets

The standard ToolTalk message sets are composed of various messages. The tables in this section show the types of messages in each of the sets:

- Desktop Services Message Set
- Document and Media Exchange Message Set
- CASE Interoperability Message Sets:
  - Analysis and Design
  - Browsing Message Set
  - Building Message Set
  - Debugging Message Set
  - Editing Message Set
  - Version Management

Desktop Services Message Set

The ToolTalk Desktop Services Message Set is composed of several request and notification messages, listed in Table 7.

Table 7 ToolTalk Desktop Services Message Set

Requests	**Notices**
Do_Command	Created
Get_Environment	Closed
Get_Geometry	Deleted
Get_Iconified	DeIconified
Get_Mapped	Iconified
Get_Status	Lowered
Get_XInfo	Mapped
Quit	Modified
Set_Environment	Moved
Set_Geometry	Opened
Set_Iconified	Raised
Set_Mapped	Reverted
Set_XInfo	Saved

Table 7 ToolTalk Desktop Services Message Set (Continued)

Requests	**Notices**
Signal	Started
	Status
	Stopped
	UnMapped

Document and Media Exchange Message Set

The ToolTalk Document and Media Exchange Message Set is composed of four request messages and one notification message, listed in Table 8.

Table 8 ToolTalk Document and Media Exchange Message Set

Requests	**Notices**
Abstract	
Deposit	
Display	
Edit	
Translate	

CASE Interoperability Message Sets

Each message set describes a different area of functionality. These are:

- Analysis and Design Message Set
- Browsing Message Set
- Building Message Set
- Debugging Message Set
- Editing Message Set
- Version Management

The Analysis and Design Message Set

The Analysis and Design message set provides an interface to functions that support software design methodologies.

Table 9 Design and Analysis Message Set

Requests	**Notifications**
AD_Attach_Object	AD_Code_Generated
AD_Check_Object	AD_Document_Generated
AD_Create_Object	AD_Edit_Centered
AD_Delete_Object	AD_Object_Attached
AD_Detach_Object	AD_Object_Checked
AD_Edit_Center	AD_Object_Created
AD_Export_Object	AD_Object_Deleted
AD_Generate_Code	AD_Object_Detached
AD_Generate_Document	AD_Object_Exported
AD_Load_Object	AD_Object_Imported
AD_Import_Object	AD_Object_Loaded
AD_Print_Object	AD_Object_Modified
AD_Rename_Object	AD_Object_Printed
AD_Save_Object	AD_Object_Renamed
	AD_Object_Saved

The Browsing Message Set

The Browsing message set provides interface functions that support cross referencing, calling relationships and class hierarchy navigation.

Table 10 Browsing Message Set

Requests	Notifications
GetLinks	
GetObject	
DisplayLibrary	
DisplayLinks	
DisplayObjects	
FocusDisplay	
ModifyObjectDisplay	

The Building Message Set

The Building message set defines the interface to a build subsystem. The build subsystem manages the processing and translation that recurs as source information is converted into a desired final product (such as an executable image).

There are two major functions the building message set performs:

1. Start a build process to create executable modules or other desired results.
2. Create and maintain dependency information that allows the build process to understand what steps need to be take to produce the desired results.

Table 11 Building Message Set

Requests	Notifications
Build	BuildFinished
CreateBuildTemplate	
StartBuild	
UpdateBuildTemplate	

The Debugging Message Set

The Debugging message set provides a control interface for a debugging utility. The messages allow another tool to ask the debugging tool to load images, control execution, and control breakpoints. This kind of interface is used when close coupling is required between an editor, debugging tools, and editor toolset.

Table 12 Debugging Message Set

Requests	**Notifications**
Assign_DebugExpression	Debug_Executing
Debug	Debug_Exited
Delete_Trap	DebugContext_Set
Describe_DebugSymbol	Debug_Resumed
Eval_DebugExpression	Debug_Stopped
Execute_Debug	Debug_Unloaded
Load_Debug	Debug_Loaded
ResumeAt_Debug	Trap_Deleted
RunUntil_Debug	Trap_Hit
UnLoad_Debug	Trap_Set
Set_Trap	DebugScope_Set
Delete_Trap	Debug_Executing
Execute_Debug	DebugStopped
Terminate_Debug	Debug_Resumed
StepInto_Debug	Debug_Exited
StepOut_Debug	Debug_Unloaded
StepOver_Debug	
Stop_Debug	
Describe_DebugSymbol	
Assign_DebugExpression	
Eval_DebugExpression	
Get_DebugScope	
Set_DebugScope	

The Editing Message Set

The Editing message set provides tight control of the editor from outside applications. These messages are generally used in situations where coupling between the editor and another tool (or tools) is required. An example of this coupling is an editor and a debugger.

The Editing message set differs from the Documentation and Multimedia message set because the editing messages provide a lower level for buffered text manipulation.

Table 13 Editing Message Set

Requests	**Notifications**
Open	
Close	
Paste	
Copy	
Set_View	
Seek	
Goto	
Annotate	
Deannotate	

The Version Management Message Set

The Version Management message set defines the interface to a version control subsystem. The messages allow integrated CASE toolsets to check in and check out elements from a version control subsystem. In addition, the version control subsystem is expected to send notifications that allow the CASE environment to react and monitor activity in the version control subsystem. This functionality can be particularly important in multiple developers or smart toolset designs where the tool builder is required to make many separate tools coordinate actions on versioned elements.

Table 14 Version Management Message Set

Requests	Notifications
CancelCheckOut	CheckedIn
CheckIn	CheckedOut
CheckOut	CheckOutCanceled
Create	Merged
Delete	
Differences	
Fetch	
GetAttributes	
GetCheckOuts	
GetContexts	
Merge	
SetAttributes	

The ToolTalk Service

A Technical Report

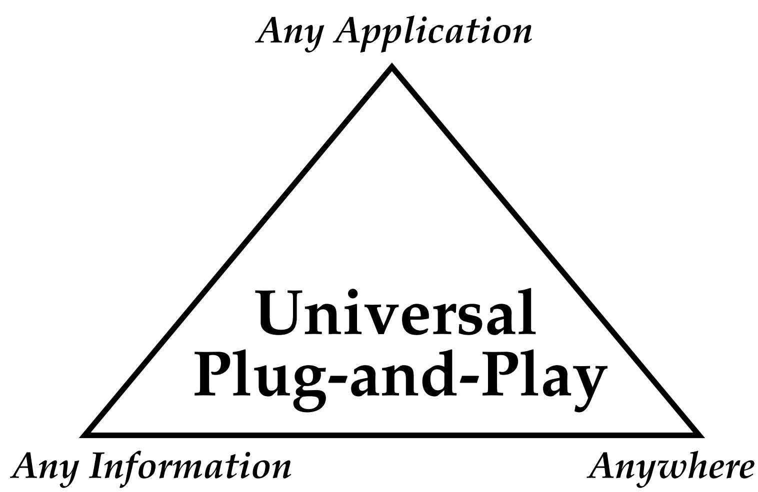


Table of Contents

SunSoft, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

For more information, call 1 800 227-9227.

Printed in USA 9/91 91022-0 1.5K