Static Message Patterns 10

10.1 Defining Static Messages

The static messaging method provides an easy way to specify the message pattern information if you want to receive a defined set of messages.

To use the static method, you define your process types and object types and compile them with the ToolTalk type compiler, `tt_type_comp`. When you declare your process type, the ToolTalk service creates message patterns based on that type. These static message patterns remain in effect until you close communication with the ToolTalk service.

10.2 Defining Process Types

Your application can still be considered a potential message receiver even when no process is running the application. To do this, you provide message patterns and instructions on how to start the application in a process type (*ptype*) file. These instructions tell the ToolTalk service to perform one of the following actions when a message is available for an application but the application is not running:

- Start the application and deliver the message
- Queue the message until the application is running
- Discard the message

To make the information available to the ToolTalk service, the ptype file is compiled with the ToolTalk type compiler, `tt_type_comp`, at application installation time.

When an application registers a ptype with the ToolTalk service, the message patterns listed in it are automatically registered, too.

Ptypes provide application information that the ToolTalk service can use when the application is not running. This information is used to start your process if necessary to receive a message or queue messages until the process starts.

A ptype begins with a process-type identifier (*ptid*). Following the ptid are:

1. An optional start string — The ToolTalk service will execute this command, if necessary, to start a process running the program.
2. Signatures — Describes the `TT_PROCEDURE`-addressed messages that the program wants to receive. Messages to be observed are described separately from messages to be handled.

Signatures

Signatures describe the messages that the program wants to receive. A signature is divided by an arrow (=>) into two parts. The first part of a signature specifies matching attribute values. 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 operation numbers (*opnum*). The ToolTalk service uses the disposition value (start, queue, or the default discard) to determine what to do with a message that matches the signature when no process is running the program. The opnum value is provided as a convenience to message receivers. When two signatures have the same operation name but different arguments, different opnums makes incoming messages easy to identify.

Creating a Ptype File

Figure 10-1 illustrates a ptype file.

```
#include "Sun_EditDemo_opnums.h"

ptype Sun_EditDemo {
             /* setenv Sun_EditDemo_HOME to install dir for the demo */
        start "${Sun_EditDemo_HOME}/edit";
        handle:
        /* edit file named in message, start editor if necessary */
        session Sun_EditDemo_edit(void)
                                => start opnum=Sun_EditDemo_EDIT;

        /* tell editor viewing file in message to save file */
        session Sun_EditDemo_save(void)
                                => opnum=Sun_EditDemo_SAVE;

        /* save file named in message to new filename */
        session Sun_EditDemo_save_as(in string new_filename)
                                => opnum=Sun_EditDemo_SAVE_AS;

        /* bring down editor viewing file in message */
        session Sun_EditDemo_close(void)
                                => opnum=Sun_EditDemo_CLOSE;
};
```

Figure 10-1 Example of a Ptype File

Figure 10-2 shows the syntax for a ptype file.

```
ptype        ::='ptype' ptid '{'
                   property*
                   ['observe:' psignature*]
                   ['handle:' psignature* ]
                   '}' [';']
property     ::=property_id value ';'
property_id  ::='start'
value        ::=string
ptid         ::=identifier
psignature   ::=[scope] op args [contextdcl]
                   ['=>'
                   ['start']['queue']
                   ['opnum='number]]
                   ';'
scope        ::='file'
             |   'session'
             |   'file_in_session'
args         ::='(' argspec {, argspec}* ')'
             |   '(void)'
             |   '()'
contextdcl   ::='context' '(' identifier {, identifier}* ')' ';'
argspec      ::=mode type name
mode         ::='in' | 'out' | 'inout'
type         ::=identifier
name         ::=identifier
```

Figure 10-2 Ptype Syntax

Property_id Information

ptid

process type identifier (ptid). Identifies the process type. A ptid must be unique for every installation. Because this identifier cannot be changed after installation time, each chosen name must be unique. For example, you can use a name that includes the trademarked name of your product or company, such as `Sun_EditDemo`. The ptid cannot exceed 32 characters and should not be one of the reserved identifiers: ptype, otype, start, opnum, queue, file, session, observe, or handle.

start
Start string for the process. If the ToolTalk service needs to start a process, it executes this command; `/bin/sh` is used as the shell.

Before executing the command, the ToolTalk service defines `TT_FILE` as an environment variable with the value of the file attribute of the message that started the application. This command runs in the environment of `ttsession`, not in the environment of the sender of the message that started the application, so any context information must be carried by message arguments or contexts.

Psignature Matching Information

scope
This pattern attribute is matched against the scope attribute in messages.

op
Operation name. This name is matched against the op attribute in messages.

Note – If you specify message signatures in both your ptype and otypes, use unique operation names in each. For example, do not specify a display operation in both your ptype and otype.

---

args
Arguments for the operation. If the args list is `(void)`, the signature matches only messages with no arguments. If the args list is empty (that is, "()"), the signature matches without regard to the arguments.

contextdcl
Context name. When a pattern with this named context is generated from the signature, it contains an empty value list.

Psignature Actions Information

start
If the psignature matches a message and no running process of this ptype has a pattern that matches the message, start a process of this ptype.

queue
If the psignature matches a message and no running process of this ptype has a pattern that matches the message, queue the message until a process of this ptype registers a pattern that matches it.

opnum
Fill in the message's opnum attribute with the specified number to enable you to identify the signature that matched the message.

When the message matches the signature, the opnum from the signature is filled into the message. Your application can then retrieve the opnum with the `tt_message_opnum` call. By giving each signature a unique opnum, you can quickly determine which signature matched the message.

You can attach a callback routine to the opnum with the `tt_ptype_opnum_callback_add` call. When the message is matched, the ToolTalk service will check for any callbacks attached to the opnum and, if any are found, run them.

In Figure 10-1, the `Sun_EditDemo_opnums.h` file defines symbolic definitions for all the opnums used by `edit.c`, allowing both the `edit.types` file and `edit.c` file to share the same definitions.

Automatically Starting a Tool

Figure 10-3 is a simple example of a ptype declaration that causes the ToolTalk service to automatically start a tool. The example code states:

> If a message to display, edit, or compose is received and there is no current instance of the tool running that can handle the messge, start "/home/toone/tools/mytest" and deliver the message.

Caution – This example causes the ToolTalk sevice to search indefinitely for a handler.

```
ptype My_Test {
    start "/home/toone/tools/mytest";
    handle:

    session Display (in Ascii text) => start;
    session Edit (inout Ascii text) => start;
    session Compose (out Ascii text) => start;

    file Display (in Ascii file_name) => start;
    file Edit (inout Ascii file_name) => start;
    file Compose (out Ascii file_name) => start;
};
```

Figure 10-3 Automatically Starting a Tool

10.3 Defining Object Types

When a message is addressed to a specific object or a type of object, the ToolTalk service must be able to determine to which application the message is to be delivered. Applications provide this information in an *object type* (*otype*). An otype names the ptype of the application that manages the object and describes message patterns that pertain to the object.

These message patterns also contain instructions that tell the ToolTalk service what to do if a message is available but the application is not running. In this case, ToolTalk performs one of the following instructions:

- Start the application and deliver the message
- Queue the message until the application is running
- Discard the message

To make the information available to the ToolTalk service, the otype file is compiled with the ToolTalk type compiler `tt_type_comp` at application installation time. When an application that manages objects registers with the ToolTalk service, it declares its ptype. When a ptype is registered, the ToolTalk service checks for otypes that mention the ptype and registers the patterns found in these otypes.

The otype for your application provides addressing information that the ToolTalk service uses when delivering object-oriented messages. The number of otypes you have, and what they represent, depends on the nature of your application. For example, a word processing application might have otypes for characters, words, paragraphs, and documents; a diagram editing application might have otypes for nodes, arcs, annotation boxes, and diagrams.

An otype begins with an object-type identifier (*otid*). Following the otid are:

1. An optional start string — ToolTalk will execute this command, if necessary, to start a process running the program.
2. Signatures — Code that defines the messages that can be addressed to objects of the type (that is, the operations that can be invoked on objects of the type).

Signatures

Signatures defines the messages that can be addressed to objects of the type. A signature is divided by an arrow (=>) into two parts. The first part of a signature define matching criteria for incoming messages. The second part of a signature defines receiver values which the ToolTalk service adds to each message that matches the first part of the signature. These values specify the ptid of the program that implements the operation and the message's scope and disposition.

Creating Otype Files

Figure 10-4 shows the syntax for an otype file.

```
otype        ::= obj_header'{' objbody* '}' [';']
obj_header   ::= 'otype' otid [':' otid+]
objbody      ::= 'observe:' osignature*
             |   'handle:' osignature*

osignature   ::= op args [contextdcl] [rhs][inherit] ';'
rhs          ::= ['=>' ptid [scope]]
                 ['start']['queue']
                 ['opnum='number]
inherit      ::= 'from' otid
args         ::= '(' argspec {, argspec}* ')'
             |   '(void)'
             |   '()'
contextdcl   ::= 'context' '(' identifier {, identifier}* ')' ';'
argspec      ::= mode type name
mode         ::= 'in' | 'out' | 'inout'
type         ::= identifier
name         ::= identifier
otid         ::= identifier
ptid         ::= identifier
```

Figure 10-4 Otype Syntax

Obj_Header Information

otid

object type identifier (otid). Identifies the object type. An otid must be unique for every installation. Because this identifier cannot be changed after installation time, each chosen name must be unique. For example, begin with the ptid of the tool that implements the otype. The otid is limited to 64 characters and should not be one of the reserved identifiers: ptype, otype, start, opnum, start, queue, file, session, observe, or handle.

Osignature Information

The object body portion of the otype definition is a list of osignatures for messages about the object that your application wants to observe and handle.

op
Operation name. This name is matched against the op attribute in messages.

Note – If you specify message signatures in both your ptype and otypes, use unique operation names in each. For example, do not specify a display operation in both your ptype and otype.

---

args
Arguments for the operation. If the args list is `(void)`, the signature matches only messages with no arguments. If the args list is empty (just "()"), the signature matches messages without regard to the arguments.

contextdcl
Context name. When a pattern with this named context is generated from the signature, it contains an empty value list.

ptid
Process type identifier for the application that manages this type of object.

opnum
Fill in the message's opnum attribute with the specified number to enable you to identify the signature that matched the message.

When the message matches the signature, the opnum from the signature is filled into the message. Your application can then retrieve the opnum with the `tt_message_opnum` call. By giving each signature a unique opnum, you can quickly determine which signature matched the message.

You can attach a callback routine to the opnum with the `tt_otype_opnum_callback_add` call. When the message is matched, the ToolTalk service will check for any callbacks attached to the opnum and, if any are found, run them.

inherit
Otypes form an inheritance hierarchy in which operations can be inherited from base types. The ToolTalk service requires the otype definer to explicitly name all inherited operations and the otype from which to inherit. This

explicit naming prevents later changes (such as adding a new level to the hierarchy, or adding new operations to base types) from unexpectedly affecting the behavior of an otype.

scope
: This pattern attribute is matched against the scope attribute in messages. It appears on the rightmost side of the arrow and is filled in by the ToolTalk service during message dispatch. This means the definer of the otype can specify the attributes instead of requiring the message sender to know how the message should be delivered.

Osignature Actions Information

start
: If the osignature matches a message and no running process of this otype has a pattern that matches the message, start a process of this otype.

queue
: If the osignature matches a message and no running process of this otype has a pattern that matches the message, queue the message until a process of this otype registers a pattern that matches it.

Figure 10-5 illustrates an otype file.

```
#include "Sun_EditDemo_opnums.h"

otype Sun_EditDemo_object {
        handle:
        /* hilite object given by objid, starts an editor if necessary */
        hilite_obj(in string objid)
              => Sun_EditDemo session start opnum=Sun_EditDemo_HILITE_OBJ;
};
```

Figure 10-5 Example of an Otype File

The `Sun_EditDemo_opnums.h` file defines symbolic definitions for all the opnums used by `edit.c`, allowing both the `edit.types` file and `edit.c` file to share the same definitions.

10

10.4 Installing Type Information

The ToolTalk Types Database makes ptype and otype information available on the host that executes the sending process, the host that executes the receiving process, and the hosts that run the sessions to which the processes are joined.

- To start applications and to queue messages, the ptype definition must be placed into the ToolTalk Types Database.
- To receive messages addressed to objects your application creates and manages, the otype definitions must also be installed in the ToolTalk Types Database.

To place your type information into the ToolTalk Types Database and make it available to the ToolTalk service, you compile your type files with the ToolTalk type compiler, `tt_type_comp`. This compiler creates ToolTalk types definitions for your type information and stores them in the ToolTalk Types Database. See Chapter 5, "Maintaining Application Information," for detailed information.

This version of the ToolTalk service provides a function to merge a compiled ToolTalk type file into the currently running `ttsession`:

```
tt_session_types_load(current_session, compiled_types_file)
```

where *current_session* is the current default ToolTalk session and *compiled_types_file* is the name of the compiled ToolTalk types file. This function adds new types and replaces existing types of the same name; other existing types remain unchanged.

10.5 Checking for Existing Process Types

The ToolTalk service provides a simple function to test if a given ptype is already registered in the current session.

```
tt_ptype_exists(const char *ptid)
```

where *ptid* is the identifier of the session to test for registration.

10.6 Declaring Process Type

Since type information is only specified once (when your application is installed), your application needs to only declare its ptype each time it starts.

To declare your ptype, use `tt_ptype_declare` during your application's ToolTalk initialization routine. The ToolTalk service will create the message patterns listed in your ptype and any otypes that reference the specified ptype.

The message patterns created when you declare your ptype exist in memory until your application exits the ToolTalk session.

Note – The message patterns created when you declare your ptype information cannot be unregistered with `tt_pattern_unregister`; however, you can unregister these patterns with `tt_ptype_undeclare`.

Figure 10-6 illustrates how a ptype is registered during a program's initialization.

```
/*
 * Initialize our ToolTalk environment.
 */
int
edit_init_tt()
{
        int     mark;
        char    *procid = tt_open();
        int     ttfd;
        void    edit_receive_tt_message();

        mark = tt_mark();

        if (tt_pointer_error(procid) != TT_OK) {
                return 0;
        }
        if (tt_ptype_declare("Sun_EditDemo") != TT_OK) {
                fprintf(stderr,"Sun_EditDemo is not an installed ptype.\n");
                return 0;
        }
        ttfd = tt_fd();
        tt_session_join(tt_default_session());
        notify_set_input_func(edit_ui_base_window,
                              (Notify_func)edit_receive_tt_message,
                              ttfd);

        /*
         * Note that without tt_mark() and tt_release(), the above
         * combination would leak storage -- tt_default_session() returns
         * a copy owned by the application, but since we don't assign the
         * pointer to a variable we could not free it explicitly.
         */

        tt_release(mark);
        return 1;
}
```

Figure 10-6 Example of a Ptype Being Registered During Initialization

10.7 Undeclaring Process Types

There may be cases when you need to retract a declared ptype; for example, in the CASE environment:

- An installation sets up a compile server which declares itself willing to accept compilation requests when it comes up. Once the server has accepted a request, it changes state and will no longer accept new compilation requests.
- A generic encapsulation process declares itself as multiple ptypes and then forwards requests to underlying tools. If an underlying tool exits, the generic wrapper no longer wants to declare itself as the ptype associated with that tool.

To unregister a ptype, use `tt_ptype_undeclare`. This call reverses the effect of the `tt_ptype_declare` call; that is, all patterns generated from the ptype are unregistered and the process is removed from the session's list of active processes with this ptype. This call returns a status of `TT_ERR_PTYPE` if the named ptype was not declared by the calling process.

Caution – One invocation of `tt_type_undeclare` will *completely* unregister the ptype regardless of how many times the process has declared the ptype; that is, multiple declarations of the ptype are the same as declaring it once.

Code Example 10-1 is an example of how to retract a a declared ptype.

Code Example 10-1 Undeclaring a Ptype

Obtain procid	`tt_open();`
Undeclare Ptype	`tt_ptype_undeclare(ptype);`

10