Thor/Theta Users Guide

Release 0.1

Barbara Liskov
Atul Adya
Miguel Castro
Eui-Suk Chung
Dorothy Curtis
Sanjay Ghemawat
Robert Gruber
Paul Johnson
Umesh Maheshwari
Andrew C. Myers
Tony Ng
Liuba Shrira
Quinton Zondervan

Programming Methodology Group Memo 95
MIT Laboratory for Computer Science
Cambridge, MA 02139

April 26, 1996

This document provides an introduction to using Thor, an object oriented database, and its language Theta. This document applies to Thor Release 0.1. Topics include installation, writing Theta code to implement objects, writing C++ application code, and debugging.

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under Grant CCR-8822158.

The authors also wish to thank all the members of the Programming Methodology Group at MIT for their help and suggestions; there were many group meetings that helped us refine the Thor design.

Contents

1 Introduction

This guide presents an overview of the Thor system and how to use it. Thor is a distributed object-oriented database system with the following features:

- Persistent storage of objects.
- Access to objects within transactions.
- Theta: A new object-oriented language for defining and implementing Thor objects.
- A multi-lingual client interface using veneers.

This document is a users guide for the Thor_0 system. This introduction presents the interface that Thor offers to the world, the architecture of the Thor_0 system, references to related documents, an outline of the rest of the guide, and system requirements for running Thor_0.

1.1 Disclaimer

Thor_0 is an experimental version of Thor that does not yet support all the intended features. It is available for experimental use only. Work on Thor continues, and more complete implementations of Thor will be released in the future.

1.2 Thor Interface

A Thor database provides its clients with a universe of objects. Each object has a type that determines its methods and their signatures. Objects in Thor are strictly encapsulated: an object can be accessed and modified only by calling valid methods on the object as specified in its type. To ensure that client programs cannot violate the encapsulation of Thor objects, client programs and Thor run in separate protection domains.

Applications interact with Thor by starting a session. Within a session an application carries out a sequence of *transactions*. During these transactions clients can modify existing Thor objects, create new objects and insert new objects persistently into Thor. Clients complete a transaction by attempting to commit the transaction or by aborting the transaction. Multiple clients can access Thor simultaneously. Thor uses optimistic concurrency control to mediate access to objects. A transaction can commit only if the objects read and written during the transaction are of the most recent version. If this was not the case (e.g., because another client has installed a new version of a read object since the transaction started), the transaction is aborted. If the commit succeeds, the transaction is serialized with respect to all other transactions. At commit time, the modifications to persistent objects are recorded reliably. When a transaction commits, a new transaction is automatically started.

The Thor universe has a persistent root object, from which all other persistent objects can be reached. An object becomes persistent automatically if it is made reachable from the root. The universe of objects is managed similarly to the heap of a strongly-typed language: unreachable objects are garbage collected.

1.3 Thor Architecture

From an implementation point of view, Thor is divided into the following components:

1. The Object Repository residing at server machine(s).
2. Front-End/Client pairs.

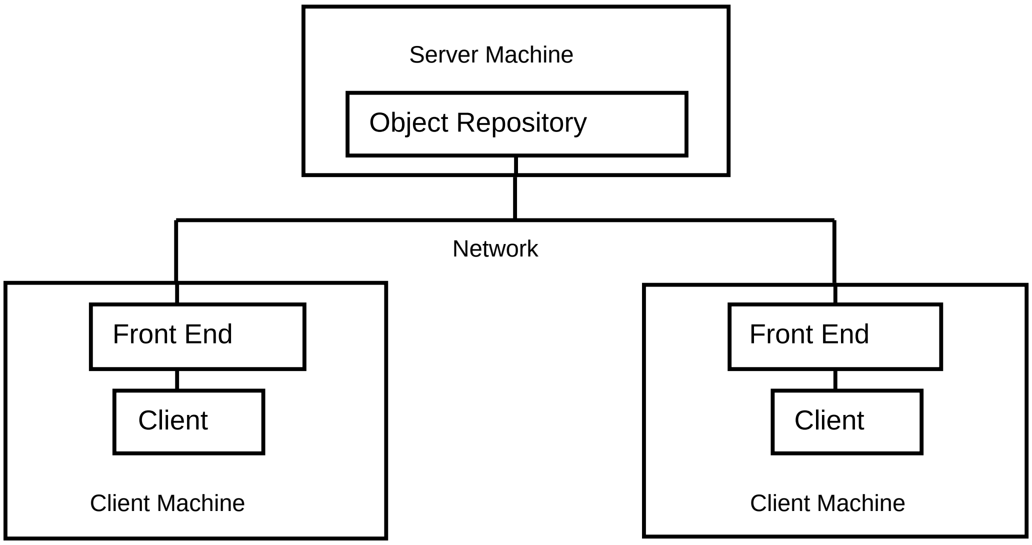

Figure 1.1: THOR

1.3.1 The Object Repository

The Object Repository (OR) is a collection of processes that manage the persistent storage of objects. A single OR can be shared by many clients. In Thor_0, the OR is implemented as a single process, started using the `or` executable, as explained in section 2. Generally, clients and ORs will execute on separate machines.

1.3.2 The Front-End

The Front-End (FE) process serves as an intermediary between the client program, and the OR. The FE has two major functions: First, the FE fetches objects from the OR as they are referenced by the client, and maintains a cache of recently used objects. Second, the FE performs method invocations on these objects on behalf of the client. To guarantee type safety, the FE type checks each call request sent to it by the client.

To enforce encapsulation of objects, and generally protect its address space from untrusted clients, the FE runs as a separate process that communicates with both the client and the OR. Communication with the client is either via shared memory or via a networking protocol (TCP/IP [11, 10]). Communication with the OR is via the networking protocol.

In Thor_0, each client interacts with a separate FE. The FE is started by running the `fe` executable, as explained in section 3. When using the network configuration, the client and FE can execute on separate machines. When using the shared memory configuration, the client and FE must execute on the same machine.

1.3.3 The Client

For every client language supported by Thor, a veneer is provided. The veneer is a small code library that implements the interface between the client and the FE. This interface consists essentially of the following operations: **connect** to an FE, **lookup** a well-known object, **invoke** methods on Thor objects, **commit** a transaction and **close** a connection. Most veneers include a library of **method stub functions**, similar to standard RPC stub functions. A method stub function is a client language function that causes the corresponding Thor method to be invoked. To allow a client program to refer to Thor objects, the FE gives out handles to objects. These handles can be used to invoke methods on the objects. Boolean, integer, real and character values are communicated directly between client and FE.

Using the corresponding veneer, a client program can be written that makes use of Thor objects during its computation. The client program first initiates a session with an FE (or starts up a new FE if desired). The client then asks for the root object of the database using the lookup command. From the root, other persistent objects can be found and accessed. When the client program terminates, it closes its connection to the FE, which terminates the FE process.

The only client language supported by the Thor_0 release is C++. However, veneers have been or are being implemented for: Java, C, Modula-3, Tcl, Erlang, and Emacs Lisp.

1.3.4 Theta

All objects in Thor are defined and implemented using a new object oriented language called **Theta** [4]. Theta is a procedural language based on CLU [8]. Like CLU, Theta supports data abstraction, exceptions and iterators. In addition, Theta supports separate type and class hierarchies, with multiple super-types, and single inheritance.

Thor includes a Theta compiler which is used to install new type and class definitions in Thor. The Theta compiler included in the Thor_0 release implements a subset of Theta, as described in Appendix B. Auxiliary routines and classes are described in Appendix C.

1.4 Related Documents

- An overview of Thor is presented in [6].
- The language-independent interface of Thor is discussed in [5].
- The Theta Reference Manual [4].
- The Theta type system is described in [7].
- A discussion of remote mobile objects in Thor is in [2].
- Garbage collection in Thor is analyzed in [9].
- The performance of Thor on the OO7 benchmark [1] is evaluated in [3].

1.5 How this Guide is Organized

The following section of this chapter, "System Requirements," describes the system requirements for Thor.

The next chapter describes the use of an Object Repository (OR). An OR provides persistent storage for objects.

The following two chapters walk the reader through the development of a small Thor application: The "FE and Theta" chapter discusses how to write Theta programs and how to install them in a Front End (FE). Theta programs define types, implementations of types (classes), and routines that create and manipulate objects stored in the Thor Object-Oriented Database. They reside in the FE, for safety reasons. This prevents errors in the client program from inadvertently changing the state of database objects.

The "Client Program and the Veneer" chapter describes the interface Thor makes available to client programs written in C++. This interface provides C++ classes that correspond to the types of the objects available in Thor.

The "Other Topics" chapter discusses the installation process, alternative configurations, and debugging.

The first appendix describes the restricted form of Theta available in this release. The second appendix contains the details of the C++ interface (veneer). The third appendix describes how the source code tree is organized.

1.6 System Requirements

This section lists the hardware and software components required to run the Thor system.

- Hardware
  - DEC Alpha station, with 64 Megabytes of memory, 100 Megabytes of disk space
  - Ethernet
  - Postscript Printer for printing documentation
- Operating System and Compilers
  - OSF/1 (aka Digital Unix) with the Software Development Environment and the Compiler Back End subsets.
  - cxx (Digital C++ compiler)
  - xdepend
  - perl
- Recommended Software.
  - tcl/tk - for using the object browser.
  - xdvi (or a postscript previewer) - for previewing documentation.
  - expect - for running test programs.
  - gm4 - for building the Theta compiler.

2 The OR

This chapter discusses the Object Repository, commonly known as the OR. An OR needs to be running before the user starts an FE or runs a client program. The user can start a new OR or use an existing OR. The OR stores the user's data persistently, either in a file or on a raw disk partition.

2.1 Starting an OR

This section assumes that a Thor distribution has been obtained and installed. Please see 6.1 for more information about installation procedures.

A pre-built OR comes with the distribution. If it needs to be built, running make in the thor/or directory builds it. The rest of this discussion assumes that an OR executable, named or, is in your path.

Starting an OR requires selecting the OR's number and a file or raw disk partition that provides for persistent storage for the OR's contents.

Each OR is identified by a name of the form <hostname>:<number>. For example, clef:0 is the OR number 0 running on the machine clef. By default, an OR created on a machine named <m> is given the name <m>:0. You can assign a different number when you create or recover the OR by using the -p flag. OR numbers range from 0 to 60956. The main requirement on the OR number is that no other OR running on a particular machine has the same OR number.

Once an OR number has been selected, for instance, 12345, set the THOR environment variable as follows:

```
setenv THOR clef:12345
```

If this OR will be reused regularly, you will want to put the setenv command in your .cshrc file.

To start an OR, use the following command:

```
or -p <number> [-s <size>] <file|disk>
```

where <number> is the OR number selected above and <file|disk> is the path-name for the file or disk partition that provides persistent storage. If you want to use a disk partition instead of a normal file, make sure that you use a raw disk partition (/dev/rrz*). Also be careful that you don't use a disk partition that contains a file system.

The file or disk that provides persistent storage will not grow beyond the size specified via the <size> option. The <size> option is specified in megabytes. (The default value is 200 megabytes.)

Example:

```
or -p 12345 /dev/rrz14g
```

To run with this OR, set the THOR environment variable to the name of the OR, e.g.,

```
setenv THOR clef:12345
```

Eventually you may be working with several different ORs. Each of these will have a different <hostname>:<number> identifier. In order for your client program to communicate with the desired OR, the THOR environment variable must be set appropriately. You may use the Unix ps command on a host to find out which ORs are running there and what their numbers are.

2.2 Shutting down the OR

The current implementation of the OR does not store its transaction log on stable storage. Therefore if the OR dies unexpectedly, it will lose committed data and the database may be corrupted. However the OR can be shut down safely by sending it SIGHUP, SIGINT, or SIGTERM. When the OR receives one of these signals it flushes all pending modifications to the disk and then exits cleanly. Therefore, the safe ways to shut down the OR are:

- Hit Control-C. This will send SIGINT.
- kill -SIGINT *OR process id*
- kill -SIGHUP *OR process id*
- kill -SIGTERM *OR process id*

Try **not** to kill the OR in the following ways: you will lose committed modifications and the database may be corrupted.

- kill -9 *OR process id*
- Crashing the machine.

2.3 Restarting the OR

Once an OR has been shut down cleanly, it can be restarted. The process is the same as starting an OR, which is described above.

2.4 Terminating an OR

Once an OR has been shut down and its data is no longer needed, its file may be removed and/or its raw disk partition may be reused.

3 The FE and Theta

This chapter describes the procedure for adding a new type to Thor. The specification and implementation of the new type are written in Theta. For details on the syntax and semantics of the Theta language, please refer to the Theta Reference Manual [4] and to Appendix B of this guide, which describes the restricted version of Theta available with this release of Thor. The first section describes how to compile Theta programs, without installing them in Thor. This is useful for becoming familiar with the Theta language. The next section describes how to install Theta types into a Thor system so that they can be used by client programs. The following section introduces an example. The following chapter discusses client programming. Debugging is discussed in section 5.

Theta code is used to describe objects that can be stored in an OR. Theta code describes the interface and implementation of these objects.

Later in this chapter, we'll see the example application Org. This application is a simple database that models an organization of workers. Within the organization, workers participate in supervisor and supervisee relationships.

3.1 Using the Theta Compiler

This section assumes that the Theta compiler and its associated OR have been installed (see Section 6.1.2) and that the theta executable is in your path. If you are using the pre-built theta compiler, you will need to set the THETA_OR environment variable appropriately. For example, if the compiler's OR is running on node gamba, type

```
setenv THETA_OR gamba:42
```

By convention, Theta source files have the extension .th. To check that a Theta file can be parsed, use

```
theta -p test.th
```

To type check your code, use

```
theta -t test.th
```

To compile your code use

```
theta -c test.th
```

To compile your code that uses types specified in another file, use

```
theta -t other-types.th -c test.th
```

This command compiles Theta code into C code that can be later linked with Thor; the code cannot be executed outside of Thor. If the compiler completes its processing correctly, you will see following output:

```
note> theta -c test.th
note>
```

When there are errors, the compiler's output is as follows:

```
note> theta -c test.th
Theta: error: test.th, line 5: syntax error
```

There are several options to specify that some source code should be emitted along with the errors. To see the line that the compiler believes is in error, use the -e1 option :

```
note> theta -e1 -c test.th
Theta: error: foo.th, line 5: syntax error
 line 5:        b: bool := any_equal(a1 a2)
```

To see the lines surrounding the line that the compiler believes is in error, use the -e2 option:

```
note> theta -e2 -c test.th
 line 1: module
 line 2: x()
 line 3:        a1: any
 line 4:        a2: any
 line 5:        b: bool := any_equal(a1 a2)
Theta: error: foo.th, line 5: syntax error
 line 6:        x: array[int] := array_new[int]()
```

3.1.1 Compiler Problems

The Theta compiler is a Thor application. This means that theta is a Thor client program and it has its own FE and OR. See section 6.1.2 for information about installing the compiler.

This section describes some situations that can occur when running the Theta compiler. These situations indicate installation problems or compiler bugs.

Compiler Accessibility

If you see

```
note> theta -c test.th
theta: Command not found.
```

it means that the theta executable is not in your path.

Compiler's FE Not Accessible

If you see

```
note> theta -c test.th
sh: cmpfe: not found
th_init failed
```

it means that the compiler can not find its FE.

Compiler's OR Not Accessible

If you see

```
note> theta -c test.th
Couldn't find root object.
th_init failed
```

it means that the OR used by the compiler is not running.

Compiler Bug

If you see

```
note> theta -c test.th
Unhandled exception: Unexpected character: '^?'; following string is: "(null)"
Internal Compiler Failure
*** Exit 255
Stop.
```

it means that you've found a compiler bug.

3.2 Building and Configuring an FE

This section assumes that a Thor distribution has been obtained and installed. Please see 6.1 for more information about the installation procedure.

A pre-built FE executable comes with the distribution. If it needs to be rebuilt, run make in the thor/fe directory.

The general Thor model is that code describing new types will be inserted into the OR by the Theta compiler and then dynamically linked into the FE, when needed. In Thor_0, code is not stored in the OR. Instead, an extra step is used to incorporate the code for the new types into a new FE.

The following sections describe the process for specifying new Theta types, implementing them and incorporating them into an FE so that they can be used by client code.

3.2.1 Adding New Theta Types to an FE

First create a file containing Theta code in the thor/fe/user_types directory. Then modify the file thor/fe/user_types/Makefile.in to add the name of the file containing the new Theta code. Finally run make in thor/fe to build a new FE with the new types.

When running a client application, the Thor veneer code will automatically find and run an FE for the application to use. All that is necessary is that the FE executable be in the user's search path.

For debugging purposes it is sometimes useful to run an FE explicitly as a separate process. Section 5 explains how to do this.

3.2.2 Changing Theta Code

During the development process, types and classes typically come and go and ORs do not hold valuable data. Later on, types and classes change less and the OR will have valuable data. Because there is a mapping between the data stored in an OR and a list of types built into the FE, it is important to preserve the mapping to ensure continued access to the data in the OR. If the mapping is lost, the data becomes uninterpretable and *no* objects in that OR will be accessible from client code. The mapping is preserved if the order of Theta files in Makefile.in in thor/fe/user_types is preserved, the order of types, classes, and routines within those file is preserved and the number and type and order of instance variables within each class is preserved. The changes that may be made that preserve this mapping are:

- Adding new files to the end of the list of existing files.
- Adding new methods to existing types.

- Adding new methods to existing classes.
- Changing the order of methods within existing types.
- Changing the order of methods within existing classes.

3.3 Example Application

The example application Org models a simplified organization of workers. Within the organization, workers participate in supervisor and supervisee relationships.

The Theta type organization describes the organization and its workers. The Theta type worker describes an individual worker, and the worker's supervisor and supervisees.

The files for this application can be found in thor/fe/user_types/example in the Thor_0 distribution:

- organization.th contains the specification for organization type.
- make_organization.th contains the specification for a creator routine for the organization type.
- organization_class.th contains the class for the type organization and the implementation of the routine make_organization.
- worker.th contains the specification for the worker type.
- make_worker.th contains the specification for a creator routine for the worker type.
- worker_class.th contains the class for the type worker and the implementation of the routine make_worker.

Here are some excerpts from the Org files. First is the organization.th file, which has Theta code describing the methods available on an object of the type organization, i.e., its specifications. % introduces a comment line.

Here is some of the contents of organization.th:

```
organization = type
  name () returns (string)
  % effects: returns the name of the organization.

  fetch (name: string) returns (worker) signals (not_found)
  % effects: returns the worker object named by name; signals not_found
  % if there is no such worker in this organization.
   ...
  all_workers () returns (vector [worker]) signals (empty)
  % effects: returns a vector containing all the  worker objects
  % in this organization.
end organization
```

The file make_organization.th contains the specification for a procedure that creates a new organization.

```
make_organization (name: string) returns (organization)
```

Part of the organization_class.th file follows. It contains code that implements the organization type. An implementation of a type is called a class. make_organization is a procedure that creates a new organization object. It is associated with a class, because it needs to know the instance variables, or the rep of the class, to initialize the new object.

```
module implements make_organization
organization_rep = class for organization
   o_name:      string implements name
   data:        directory

   fetch (wname: string) returns (worker) signals (not_found)
      ...
      end fetch
      ....
   all_workers () returns (vector[worker]) signals (empty)
       ....
       end all_workers
end organization_rep

make_organization (name: string) returns (organization)
   return (organization_rep{
            o_name  := name,
            data    := create_directory()
      })
end make_organization

end % module
```

3.3.1 Putting the Example Application Code into an FE

The Theta compiler needs to have been installed and an OR for the compiler needs to be running in order to compile and incorporate Org's Theta code into Thor. See section 6.1.2 on how to install the compiler and start its OR.

To compile Org's Theta code:

1. copy the files containing Org's Theta code from thor/fe/user_types/example into the directory thor/fe/user_types.

   The following files should now be in thor/fe/user_types:

   - Makefile.in
   - dummy.th
   - organization.th
   - make_organization.th
   - organization_class.th
   - worker.th
   - make_worker.th
   - worker_class.th

2. modify the section THETA_SOURCES in the file Makefile.in to name the files containing the user Theta code.

   The file Makefile.in should now contain:

```
TOP = ../..
srcdir = @srcdir@
VPATH = @srcdir@

  ...

THETA_SOURCES = \
        $(srcdir)/dummy.th \
        $(srcdir)/organization.th \
        $(srcdir)/organization_class.th \
        $(srcdir)/make_organization.th \
        $(srcdir)/worker.th \
        $(srcdir)/worker_class.th \
        $(srcdir)/make_worker.th

CCINCS = -I. -I$(srcdir) -I$(srcdir)/.. -I$(srcdir)/../.. -I$(TOP) \
                -Itheta_generated -I../types -I../types-th/theta_generated
CCCINCS = $(CCINCS)

  ...

THISLIB = user_types.a
CONDITIONAL_BUILD = USER
include $(TOPSRC)/config/libdefs
```

   Note that only the THETA_SOURCES line need be changed.

3. To develop, just run make in thor/fe/user_types. This will compile the user's Theta code against existing persistent Theta code without making the new code persistent.

4. To install the new types in the FE, run make in thor/fe subdirectory. This will make the new a new FE that incorporates the types, classes and routines described in the user's Theta code. This FE will make the new types and routines available for use by client applications.

4 Client Programs and the Thor Veneer

This chapter presents an overview of writing a C++ client application. It then continues with the Org example begun in the previous chapter and shows the building of the client program for the organization example. The last section describes how to build a client on a Sparc workstation.

4.1 Writing a Client Application

This section provides an overview of writing a C++ client application for Thor. The restrictions on the veneer are discussed in Appendix B.

C++ applications use the objects and routines stored in Thor through an automatically generated library called the *veneer*. There are three classes of objects and routines in the veneer. There are utility routines for connecting with an FE, built-in types and routines, such as int and th_get_root, and user-defined types and routines. This chapter gives an overview of the available functions in the C++ veneer. The exact specifications of the utility functions are present in the file thor/client/C++/veneer/thor.h. All functions available in that file are public; it is advisable for the application programmer to avoid using functions declared in other header files. Thor routines have a prefix th_; the application programmer should avoid using this prefix for variables, functions, etc.

A C++ application can use Thor objects after starting a "session" with the Thor universe. The procedure th_init is used to start a session. All calls to Thor must be done after this call. When an application is done with its work, it closes its session with Thor by calling th_shutdown. No more calls to Thor may be made after this call.

4.1.1 Objects and Routines

Objects of the types null, bool, char, and int are passed between the application and Thor by value. They are called primitive types. Their C++ counterparts are null, bool, char, and int. The Theta literals true and false are available as TRUE and FALSE. They are defined consistently with the standard C++ idiom of 0 for FALSE and 1 for TRUE and can be used whenever a boolean literal is needed. The special literal th_nil can be used to name the Theta literal nil.

Other types, including all user-defined types, are passed as opaque objects. Objects of a Theta type named foo have the C++ type th_foo. The type and its methods are declared in the header file th_foo.h.

A Theta routine bar is available to applications as th_bar. The function prototype is declared in th_bar.h.

For example,

```
#include "th_stack.h"
#include "th_stack_create.h"
th_init();
th_stack s = th_stack_create();
s.push(17);
int i = s.pop();
```

4.1.2 Case Sensitivity

The C++ programmer should be aware of the case sensitivity issue in mapping Theta names to C++ types: C++ is case sensitive but Theta is not. Theta names are canonicalized by converting all letters to lower case. Therefore, the derived C++ names are all in lower case as well. For example, if a Theta type is named ExampleType, the corresponding C++ name would be th_exampletype.

4.1.3 Exceptions

A method or routine can raise an exception listed in its Theta specification. It can also raise the special exception failure, indicating some failure within Thor, or abort, indicating that the current transaction was forced to abort.

In later releases Thor exceptions will be mapped to C++ exceptions. Currently, the following macros are used to determine if a Thor exception has been raised. The routines available for catching exceptions are th_catch and th_catch_any. The former allows an application to handle a particular exception while the latter is used for handling any exception. The function th_raise can be used by an application for raising an exception; the effect is as if the exception had been raised by Thor. A routine th_assert_no_exceptions allows an application to assert that no unhandled exceptions, other than abort, are outstanding.

After an exception has been raised and before it has been handled, all invocations to Thor are ignored, i.e., each invocation simply returns an invalid object. A special invalid object is returned in case of a non-primitive type. It is the application programmer's responsibility not to invoke methods on an invalid object; otherwise, the program will crash. For primitive types, special values are returned in the case where an exception has not yet been handled: 0 for integer/character/null, FALSE for boolean. Where a C++ string (char*) is expected (e.g., th_string_to_chars), the empty string is returned.

4.1.4 Miscellaneous Routines

The following routines are available to the application programmer:

th_init — Connect to Thor. This routine must be called before any other Thor routine.

th_shutdown — Closes the session with Thor. No Thor routine should be invoked after this call.

th_get_root — Returns the root object of the Thor universe.

th_equal — Used for testing whether two objects are the same

th_is_invalid — Used for checking whether a (non-primitive) object is the special invalid Thor object.

th_disable_futures — Facilitates debugging by making the client interactions with the FE synchronous. This makes finding the cause of an unhandled exception easier.

th_string_to_chars — Converts a Thor string to a null-terminated C++ string (char*).

th_chars_to_string — Converts a null-terminated C++ string to a Thor string.

4.1.5 Transactions

Each method invocation is invoked as part of a transaction (also referred to as the current transaction). When the application wants to make its changes persistent or verify that the work done so far as part of the current transaction is serializable, it can commit the transaction using th_commit. At this time, if there is an unhandled exception other than abort, the veneer crashes the program. If the transaction commits, all the modifications to persistent objects become permanent. Furthermore, all non-persistent objects that are reachable from these modified objects are made persistent.

The current transaction's work can discarded by using the th_abort function. All changes to persistent objects are undone. Objects created during this aborted transaction are reverted to their initial state (i.e., their state at creation time). If there is any unhandled exception at the time of the th_abort call, the veneer clears it. After the completion of either of th_commit or th_abort, a new transaction is started, i.e., all subsequent calls to Thor are part of the next transaction.

4.1.6 Subtyping

For non-primitive Theta types, if type foo is a subtype of bar, the C++ class th_foo will be a derived class of th_bar.

For example, given that all Theta types are subtypes of any, it is valid to pass a th_project where a th_any is expected:

```
th_directory root = th_get_root();
th_string s = th_chars_to_string("project");
th_project p = th_project_create();
root.insert(s, p);
```

Using Anys

All Theta types including primitive types (null, bool, char, int) are subtypes of the Theta type any. A non-primitive object can be used wherever an object of type any is expected. However, primitive objects need to be explicitly converted to an any object using a routine of the form th_<primitive>_to_any. For example:

```
th_directory root = th_get_root();
th_string s = th_char_to_string("project_id");
root.insert(s, th_int_to_any(17));
```

Casting to a Subtype

The th_force macro can be used to force the type of any Thor object to a specific non-primitive type:

```
<th_type> th_force(th_any x, <th_type>)
// effects    If the real type of "x" is a sub−type of "<th_type>",
//            return "x" with type "<th_type>".
//            Else raise the exception "wrong_type" and return an
//            invalid object of type "<th_type>".
```

For example,

```
th_any a = root.lookup(th_chars_to_string("project"));
th_project p = th_force(a, th_project);
```

For a primitive type, the routines th_force_to_<primitive> can be used to cast an object of type any to a primitive object. These routines can signal wrong_type if the actual type of the object is not the same as the expected primitive type.

4.1.7 Parameterized Types

The C++ veneer supports the Theta parameterized types array, maybe, and vector.

Parameterized types are implemented in the veneer using C++ templates. The syntax for array[foo], maybe[foo] and vector[foo] is:

```
th_array<th_foo>    af;    // Array of foo
th_maybe<th_foo>  mf;    // Maybe of foo
th_vector<th_foo> vf;     // Vector of foo
```

Note that with nested parameterized types it is important to add appropriate white space so to avoid syntactic confusion with the C++ >> operator, e.g.:

```
th_array<th_maybe<th_foo> > amf;    // Array of maybe of foo
```

To use parameterized types, a C++ program must have the following as the last Thor #include line:

```
#include "th_instantiations.h"
```

This file contains a #pragma define_template line for each instantiation of a parameterized type in the veneer.

4.1.8 Developing a Thor Client

To begin the client part of a new Thor application, use the subdirectory thor/client/C++/user. Put the client C++ code in a file in this directory. This code can use Thor veneer routines specified in thor/client/C++/veneer/thor.h and in the include files in thor/client/C++/veneer that describe the new types that the user has incorporated into the FE. Makefile.in is edited to include the new C++ files. make is then run in the user directory to compile and build the application. If an OR is running and the THOR environment variable is defined correctly, running the client application will cause an FE to be started which will contact the OR and give the application access to the Thor objects stored there.

4.2 Example Application (Continued)

There are three steps left in creating the sample Thor application Org: creating the veneer for the new types, building the application itself and then running the application.

4.2.1 Creating the Veneer

Typing make in the thor/client/C++ directory will create include files for all the types that the FE knows about. These include files will be in the thor/client/C++/veneer directory. In the case of the Org example, the files th_organization.h, th_make_organization.h, th_worker.h and th_make_worker.h, will be created, in addition to files for the various built-in types.

4.2.2 Building an Application

The file orgtest.cc contains the client C++ code for Org. In the Thor$_0$ release it can be found in thor/client/C++/example.

Below are several fragments from the file orgtest.cc.

Header files included in COMMON section are required by all C++ applications. Header files included in DIRECTORY section allow access to objects of Theta type directory. Header files included in ORG section allow access to Org's Theta code. For example, including in the client code a header file th_organization.h allows the client to invoke methods of Theta type organization. The header file th_instantiations.h, that must be included as the last Thor #include files, allows access to Theta parameterized types.

```
// COMMON
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>

#include "common/th_assert.h"
#include "thor.h"
#include "th_Class.h"
#include "th_string.h"

// DIRECTORY

#include "th_directory.h"


// ORG

#include "th_organization.h"
#include "th_make_organization.h"
#include "th_worker.h"

#include "th_instantiations.h"

.....

void main(int argc, char **argv)
{
   th_organization org;
   char const* exc;

   // Open connection with FE
   th_init ();
   disable_futures();

   th_string project_name = th_chars_to_string("Org");

   // Get the root of the initial OR.
   th_directory root_dir = th_get_root();

   if (th_catch_any(exc)) {
      fprintf(stderr, "Exception: %s\n", exc);
```

```
        exit(EXIT_FAILURE);
    }


    ....

    th_commit();
    if (!th_catch("abort"))
        cerr << "Transaction Committed\n";
    else
        cerr << "Transaction Aborted\n";

    cout << "Exiting\n";

    th_shutdown();

}
```

To create the Org application do the following:

1. Start in the thor/client/C++ directory.
2. Copy the file orgtest.cc containing Org's C++ client code from the example subdirectory into the user subdirectory.
3. In thor/client/C++/user edit Makefile.in to add orgtest.cc to the CCCSRCS line and add an appropriate make entry for orgtest:

   ```
   orgtest: orgtest.o $(DEPS)
           $(CCCLINKER) -o $@ orgtest.o  $(LIBS)
   ```

4. Run make in thor/client/C++.

   The subdirectory thor/client/C++/user now contains files

   - Makefile
   - Makefile.in
   - orgtest.cc
   - dummy.cc

5. In user, type make orgtest to compile and build the orgtest application.

4.2.3 Running the Org Application

1. Select a number for the OR and define the THOR environment variable. (See section 2 for more information on running an OR.)

   setenv THOR foohost:1234

2. Run an OR on host named foohost:

   or -i -p 1234 /tmp/orgor

3. Make sure the fe executable is in your search path and run orgtest.

4.3 Running clients on a SUN Sparc

The current version of Thor runs only on DEC Alpha stations, i.e., the Thor servers must always reside on Alphas. Client programs, however, can run either on Alphas or on SUN Sparc stations. This section describes the steps needed to run client programs remotely on Sparcs.

At this time, only C++ clients are supported in this mode. The client program must be compiled and linked with the Thor C++ veneer and stub code for those types used by the client program.

4.3.1 Compiling Your Client

The following portions of the Thor source code must be copied to the Sparc station and compiled and linked with the client:

- C++ veneer code
- C++ stubs for all types used by the client program.
- Some additional files used by above mentioned code.

The Makefile in directory thor/client/C++/sun_tar has a target called suntar that produces a tar file containing all the source files mentioned above. The tar file is called thor-sun.tar.gz. Please review the options to the tar command in the Makefile and below to make sure they are appropriate for your system.

Follow these steps to install the code and compile your client programs:

1. Copy thor-sun.tar.gz to a directory where you want to install it.
2. Uncompress the file using the command : gunzip thor-sun.tar.gz.
3. Expand the file using the command : tar xvof thor-sun.tar. This gives you a directory called thor.
4. Go to the directory thor, where you will find a script file called configure. Execute that script. This script creates a file called config/config.h. (On a DEC Alpha it also creates a many Makefiles, but on a SUN it doesn't. Please ignore all messages related to Makefiles.)
5. For convenience, you can put the following lines in your Makefile for the client program.

```
THOR_DIR= ~/thor
THOR_VENEER_DIR= ~/thor/client/C++/veneer

THOR_OBJECTS =\
   $(THOR_DIR)/common/hostname.o \
   $(THOR_DIR)/common/intarray.o \
   $(THOR_VENEER_DIR)/binary_veneer.o \
   $(THOR_VENEER_DIR)/binary_veneer_remote.o \
   $(THOR_VENEER_DIR)/th_classes.o \
   $(THOR_VENEER_DIR)/reference.o \
   $(THOR_VENEER_DIR)/vtable.o
```

   And finally, compile the files in THOR_OBJECTS and link them with your application program. Depending on the compiler you use, you might experience some problems with files referenced in #include directive. Please modify them to make them work in your environment.

4.3.2 Running Your Client

When running client programs remotely on Sparcs, they can not automatically create an FE on the server machine. You can only connect your client programs to existing FEs.

In order to do this, find out your FE's location and port number, and set the environment variable FE_LOCATION to it. The following shell command will do it assuming that your FE is running on gamba.lcs.mit.edu on port 4225 :

```
setenv FE_LOCATION gamba.lcs.mit.edu:4225
```

When your client program invokes th_init(), it will connect your client to the FE specified in FE_LOCATION.

Also, because this connection will be made via TCP, rather than shared memory, it requires that the FE be built with the SHM option set to FALSE in thor/config/vdefs.in. Then type make depend and make in thor.

5 Debugging

This section assumes that the reader is familiar with debugging C and C++ code with **ladbx** or **decladebug**. This section presents some clues about debugging Thor applications. First the relevant architecture and the conceptually easy debugging approaches are presented. The final section discusses how to debug more serious problems in an FE.

5.1 Where is the bug?

Under normal conditions, running the client program automatically starts up an FE. Thus it is not apparent to the unsuspecting user that two processes are running on his behalf: the client process and the FE. If the client program apparently misbehaves, the bug may be in the user-written client code, the user-written Thor types, or elsewhere.

When running a client program, if the message FE crashed is displayed, then the fault is most likely in the Thor types (or the FE). If the FE crashed message is not displayed, then the problem is most likely in the client code.

One common cause of client code crashes is omitting the call to th_init at the very beginning of the client program.

Another common client code problem is ignoring exceptions.

5.2 Debugging Client Code

When debugging the client code, there is a choice is between print statements and using a debugger. Given that the user is familiar with the client code programming environment, the user can make the best choice between the two options.

5.3 Debugging the FE Code

The conceptually easy way to debug Thor types and the FE code is to insert calls to the print routine at judicious points. It may be important to put a newline character "\n" at the end of the string being printed out, both for readability and to encourage Unix to print it promptly.

At times print messages are insufficient to debug a problem and more powerful tools are necessary. The basic approach here is to run the FE under a debugger. A typical sequence is as follows:

```
cd thor/fe
ladbx fe
stop in complex_get_x
run -P 4000
```

This sequence starts up a debugger for the FE, sets a breakpoint in the complex_get_x method of the type complex, and starts the FE running with a specific shared memory segment (4100). The mapping of Theta names to C names is discussed in section B.1.5.

Now that the FE is started, one needs to start the client and have it communicate with the FE being debugged. The following sequence accomplishes this goal:

```
cd thor/client/C++/user_dir
setenv FE_LOCATION 4100
```

user_pgm

This sequence sets the environment variable FE_LOCATION to specify the name of the shared memory segment that is being used by the FE that is being debugged. Of course, the client program may be run under a debugger at the same time as the FE is being debugged.

Note that if TCP/IP communication is being used, the run command to the FE being debugged needs to specify a host and a port number on which the FE is listening for client communications: run -P <hostname>:4100. Also, for the client program to find the FE, the setenv command needs to be changed too: setenv FE_LOCATION <hostname>:4100. In this context port numbers range from 256 to 65535.

To revert to having the FE automatically started by the client, type unsetenv FE_LOCATION.

5.4 Releasing Shared Memory Segments

Occasionally, when the FE terminates abnormally, it does not free the shared memory segments it is using. Use the ipcs -a command to see if there are shared memory segments in use. Use the ipcrm command to free shared memory segments.

6 Installation and Configuration

6.1 Installation

The installation of Thor involves retrieving the Thor distribution from ftp://ftp.lcs.mit.edu:/pub/thor, unpacking the tar file, installing the compiler and building the Thor system.

The Thor installation supports two configuration choices: an integrated build tree and a separate build tree. The integrated build tree contains both sources and generated files. By choosing a separate build tree, the sources are separated from the generated files. There are several reasons for choosing a separate build tree:

- To maintain several different collections of generated files, e.g., one build tree for debugging code, one for optimized code.
- To put re-creatable files on non-backed up storage.

The main reason for choosing an integrated tree is simplicity. The user creates code and invokes make in the same directory hierarchy. The examples in this guide can be followed exactly. It is less likely that you will type make in the wrong directory or invoke the configure command incorrectly.

Some additional information in making this choice is the following: the fully installed Thor system uses about 70 Megabytes of disk space. About 35 Megabytes of this is generated code.

Please note that when switching from an integrated build tree to a separate build tree, it is important to do make clobber in the integrated build tree, before running make in the separate build tree.

6.1.1 Initial Installation Steps

Here is the procedure to begin installing Thor:

1. Create the directory that will contain the Thor distribution.
2. Obtain the compressed tar file: ftp://ftp.lcs.mit.edu:/pub/thor/thor0.0.tar.gz.
3. Uncompress and untar the compressed tar file.
4. Install the compiler.

6.1.2 Compiler Installation

There are two choices in installing the compiler. Either the directory containing the compiler's pre-built executables can be put in each user's path or the executables can be installed (via links or copying) in a directory that is already in users' paths. After this, the compiler's OR should be started. Since the compiler's OR is expected to be long running, it may be useful to update your machine's reboot procedures to automatically restart the compiler's OR. The directory cmp-exe contains the pre-built executables for the compiler: theta, cmpfe, and cmpor. This directory also contains a script to start the compiler's OR, named start_cmp_or. You will need to edit this file to reflect your installation. If the pre-built compiler is installed, users will need to set an environment variable, THETA_OR, to reflect the location of the compiler's OR:

```
setenv THETA_OR gamba:42
```

6.1.3 Final Installation Steps

To continue installation with a separate build tree, perform the following steps:

1. Create the directory to contain the separate build tree.

2. Change to that directory.

3. Type the full path name of the directory containing the thor sources followed by /configure. For example, /nfs/gamba/u/joeuser/thor/configure. Use of relative path names, such as those beginning with ../ or thor/ will result in Makefiles that don't work.

To complete the integrated build tree installation, perform the following step:

1. Type configure.

The next step for either type of build tree is to answer default to configure's question, "What configuration do you want to build?" configure will then configure the default system. When the configure phase is complete, type make boot.

6.1.4 Executables

At the end of the installation process, several important executables have been built:

- The OR executable is in the thor/or subdirectory. It is named or.

- The FE executable is in the thor/fe subdirectory. It is named fe.

Once built, the OR executable is expected to be fairly stable and can be started once and left in the background. During development, each user will probably be using a different OR.

As types are created, the FE executable changes. Client programs usually start the FE automatically. Thus, fe both changes from user to user and needs to be in the user's path. On Unix, there are many ways to achieve this:

- Add a link fe to the user's bin directory.

- Use an alias.

- Put a link fe executable into the relevant thor/client/C++ directory.

6.1.5 Installation Problems?

If the pre-built compiler doesn't work, the compiler can be rebuilt from the distribution. In doing so, the installer may wish to redefine PUBLIC_THETA_OR in thor/client/C++/theta_compiler/theta.h to reflect the machine that will be running the compiler's OR. After this, the use of the environment variable THETA_OR will no longer be necessary. Building the compiler is accomplished by using configure, selecting the compiler configuration, and typing make boot. When the build is complete, make save-cmp will overwrite the compiler executables in the thor/cmp-exe directory.

If you are having problems with the installation or other aspects of thor, please send e-mail to thor@lcs.mit.edu.

6.2 Configuration

6.2.1 Default and Optimized

The default configuration of the Thor source tree is to build the OR and the FE with many assertion checks to validate the implementation. User Theta code will also be generated with such (internal) checking. Once code has been debugged you can rebuild the OR and FE with checking turned off and optimization turned on. To do this run configure in the Thor root directory and answer optimized to the question "What version of Thor do you want to build?." If you have an OR running with state that you wish to preserve you should kill it safely (see section 2.2) and restart with your new OR.

6.2.2 Other Configuration Options

The file thor/config/vdefs.in contains many detailed options available in building Thor. It is not expected that many users will be interested in most of the options in this file. However, in order to use the Emacs interface or the Tcl Object Browser or to use a client program on a Sparc station, it is necessary to build Thor to use TCP/IP sockets, rather than shared memory. This is accomplished by changing the value for the SHM option in thor/config/vdefs.in from TRUE to FALSE. Then run make depend and make in thor.

A Performance

This appendix outlines the steps required to build a version of Thor suitable for running the OO7 database [1] benchmarks and obtaining good performance. The performance obtainable with this release is not as good as that reported in paper [3] on hot traversals, but is otherwise similar.

A.1 Building

To build the oo7 configuration with compiler optimizations enabled,

```
cd thor
make clobber
configure -with-CCDEBUG=O oo7_th
```

This builds a version of the Thor system with only the OO7 data types and with client-FE communication via shared memory.

A.2 Creating the Environment

To run the OO7 benchmark, first start the OR and then setup the appropriate environment for the FE.

```
setenv THOR <hostname:number>
or -i -p number <file_name>
setenv FE_FLAGS "-s 26000"
```

These flags cause the FE to run with a 26 Megabyte cache so that no garbage collection occurs.

A.3 Running the Benchmark

Running the benchmark involves running client code to create a database and then running client code to traverse the database. The following commands create a small database named sm_db:

```
cd thor/client/C++/oo7
oo7
s       --- create a small database
sm_db   --- a name for the small database
q       --- quit
```

The sizes of databases that can be created are

```
v - very small
s - small   (7.8 Megabytes)
m - medium (72.9 Megabytes)
b - big
```

The following commands traverse the database and print out the resulting times.

```
cd thor/client/C++/oo7
oo7
T      --- select a traversal type
sm_db  --- name of the small database
2b     --- select a particular OO7 traversal kind
h      --- select a hot traversal
m      --- run each traversal as a separate transaction
q      --- quit
```

The types of traversals are

```
t - run the traversal inside the client
T - run the traversal inside Thor
M - run the composite part traversal inside Thor
```

The kinds of traversals are

```
1  --- Raw traversal, read-only, visit every part.
2a --- Update one atomic part per composite part.
2b --- Update every atomic part as it is encountered.
2c --- Update each atomic part in a composite part four times.
6  --- Sparse traversal: like 1, but visit only the root part of each atomic part.
10 --- Switches between 1 and 6 on even/odd composite parts.
```

Either a hot or cold traversal may be selected. A hot traversal will actually traverse the database 5 times, the first one is used to warm up the cache. The time shown at the top of the output is the average of the middle 3 traversals. The cold traversal traverses the database once.

B Restrictions

This appendix describes the restrictions on Theta in the Thor_0 release and on the C++ veneer.

B.1 Theta Restrictions

The Theta Reference Manual [4] describes the full Theta language. This section describes the subset of Theta available in the Thor_0 release.

B.1.1 Subtyping and Subclassing

Types may be subtypes of at most one other type. There is no multiple subtyping. Renaming in type definitions is not supported.

There is no support for subclassing (also known as inheritance). Therefore renaming in subclass definitions is not supported, nor is there support for the hides and provides clauses, nor is there support for makers or the make statement.

B.1.2 Equates, Modules, Types, and Classes

Type equates are supported, but expression equates are limited to literals.

In the implements clause on a module header, renaming is not supported.

User-defined parameterized types are not supported. The built-in types array and vector are available.

The keyword const_ivars is allowed in a class header to tell the compiler to assume that the state of an object of this class does not change after creation. The form of use is

```
x = const_ivars class
```

Class constructors are supported. Multiple implementations are supported.

B.1.3 Procedures and Iterators

Routines that return multiple results are not supported. Multiple assignment statements are not supported. Varying argument lists are not supported. Closures are not supported. Values are not supported in signal statements.

The following built-in iterators are supported: to (for int), indexes (for vector and array), elements (for vector). The following built-in iterators are *not* supported: to_by (for int), elements (for array).

User-defined iterators are not supported.

B.1.4 Built-in Environment

The supported built-in types are any, null, int, bool, char and string. The built-in parameterized types array, vector, and maybe are supported, but without where clauses. Thus the following methods are not supported for arrays, vectors and maybes: copy, similar, unparse. The following constructors are not supported: array_create, array_generate, vector_create, vector_generate. The following built-in types are *not* supported: real, sequence, record, struct, oneof, routine.

The use of uninitialized variables is not detected.

B.1.5 Names

Due to the relatively straight-forward way in which the Theta compiler currently translates Theta into C, various name collision problems can occur. Users select names for various entities in Theta, such as types, classes, routines, local variables, and formal parameters. The compiler then uses these names, as well as some derived names. Further this C code is linked with the FE and with some Unix libraries, both of which contain names that can conflict with names chosen by the user. This subsection discusses which conflicts can happen.

When a user chooses a name for a class, for example, small with a method size, the compiler generates the following derived names: initsmall, small_methods, small_size.

When a user chooses a name for a routine, for instance time, the compiler generates the following names: inittime_rtn_class, time_classdv_s, time_rtn_class_methods.

If a user has several modules, the names of routines in these modules should be distinct.

The following is a short list of common C subroutines that are usually linked into the FE, whose names the user should avoid (this list is incomplete): errno, open, read, write, close, exit, free, index, printf, time, wait, sleep.

The user should also avoid keywords in the C language. For instance declaring a local variable (or a formal, or a routine name) "static" won't work at this time.

The following names are typical of names used in the FE, with which the user should avoid conflicts: any_as_obj, array_append, binaryDispatch, cache_fetch.

B.2 C++ Veneer Restrictions

The C++ Veneer in general provides support for only those Theta features described in section B.1. The C++ programmer should be aware of the following restrictions on the C++ veneer:

1. No real numbers.
2. No support for iterators.
3. No support for multiple super-types.
4. No support for user-defined parameterized types.
5. No support for parameterized routines.
6. No support for where clauses for built-in parameterized types.

C Library Classes and Routines

This appendix describes the standard library. The types and routines defined in this library can be used both in Theta and in client applications. See chapter 4 for information on how to use the standard library from client applications.

C.1 Directory Type

A directory is a map from names (strings) to objects (anys). The directory type provides the following methods:

```
lookup (name: string) returns(any) signals (not_found)
    % effects: returns the object named by name; signals
    % not_found if there is no such object in the directory.

insert (name: string, obj: any) signals (exists)
    % effects: inserts object obj named by name in the directory;
    % signals exists if there is already an object named by name in
    % the directory.

remove (name: string) signals (not_found)
    % effects: removes the object named by name;
    % signals not_found if there is no such object in the directory.

size () returns (int)
    % effects: returns the number of objects in the directory

names ( ) returns (directory_gen)
    % effects: returns a generator that can be used to obtain
    % the names in the directory.
```

The type directory_gen provides the method next described below. The user can generate all the names in the directory by calling this method repeatedly until it signals done. The order in which names are generated is undefined. The directory whose elements are being generated, should not be modified while the generator is being used.

```
next ( ) returns (string) signals (done)
    % effects: returns the next name in directory;
    %signals done when there are no more names.
```

C.2 Directory Routines

The create_directory routine creates an object of type directory. The current implementation is very simple and inefficient: lookup, insert and remove are all $O(n)$.

```
create_directory () returns (directory)
    % effects: creates a directory.
```

C.3 Persistent Root

Each OR has a persistent root object. This object has type directory and it can be retrieved using the method get_root. Since Thor offers persistence by reachability, the root object can be used to make objects persistent.

get_root () **returns** (directory)
% **effects**: *returns the root directory for the default OR*

C.4 Local Name Space

Each execution of a program gets its own local directory object, initially empty. More precisely, each FE has a private directory object which is created when it boots. The lifetime of the object is the lifetime of the FE, unless the object becomes persistent. This object is returned by the routine local_directory. Applications can use this local name space for storing pointer(s) to persistent data that is used across several modules.

local_directory () **returns** (directory)
% **effects**: *returns the local directory for the FE.*

C.5 Print

The print routine outputs a string to *stderr*. It is mainly useful for debugging.

print (msg: string)
% **effects**: *Prints the message msg to stderr.*

D Overview of Source Tree

This section is intended to give users an idea of the contents of the source file directory hierarchy. This is not a complete description of the source code, but it allows interested readers to find the code they want more easily.

bin	Various administrative shell scripts for Thor.
client	Contains various application and test programs that use Thor. It is divided into subdirectories by programming language. The languages include C, C++, Erlang, Emacs Lisp, Perl, Modula-3, Tcl.
client/C++/OO7	Implementation of OO7 benchmark suite.
client/C++/example	Sample C++ client code for organization example.
client/C++/sun_tar	Contains a makefile that generates a tar file that lets users run clients on Sun Sparc workstations.
client/C++/ theta_compiler	Implementation of Theta compiler user interface.
client/C++/user	Directory designated for the creation of new C++ clients.
client/C++/veneer	C++ veneer runtime system, with support for futures and promises.
client/tcl	A graphical object browser.
cmp-exe	Contains pre-built compiler executables.
common	Common routines and data structures shared by various pieces of Thor. In particular, the FE and the OR depend on the contents of this directory.
config	Contains configuration information for Thor for compilation. Some of the files in this directory and all files in config/vdefs are generated automatically.
config/vdefs	Automatically generated C++ header files based on Thor configuration.
doc	Documentation of Thor.
fe	Implementation of front end (FE).
fe/boot	Reads in the Theta type specifications and builds internal type information during FE start up. Also contains the type specifications for Theta built-in types.

fe/cache	Implements FE cache, garbage collection and pointer swizzling.
fe/client	Veneer stub generators for various languages, FE interface to veneers, batched futures and control structure implementation.
fe/compiler_type	Implementation of Theta compiler.
fe/inquir_types-th	Type specifications and implementation in Theta for Inquir database.
fe/labbase_types	Type specifications and implementation for Labbase.
fe/oo7types-th	Type specifications and implementation in Theta for OO7 benchmarks.
fe/runtime	Various runtime routines for FE, including object invalidation, exception handling, surrogates, pointer unswizzling, object dispatch header installation and type casts.
fe/tests-th	Type specifications and implementation for Thor tests.
fe/types-th	Basic Thor type and routine specifications and implementation.
fe/types	Theta built-in type implementations, such as `array` and `int`. (The specifications can be found in fe/boot).
fe/user_types	Additional type definitions written by the user.
fe/user_types/ example	Sample Theta code for organization example.
or	Implementation of object repository (OR). This directory includes transaction manager, FE and OR connection managers.
or/mm	Implementation of OR memory manager, log manager and disk I/O routines.
or/mm/lmm	A quick log-structured implementation of the memory manager (as opposed to in-place). It overrides various pieces of code implemented in the or/mm directory.
pre-built	Contains pre-built or and fe executables.
tests	Various test code files for the Thor code.

E Specifications for Client C++ Interface

```
/* Copyright Barbara Liskov, MIT, 1996 */

#ifndef _THOR_H
#define _THOR_H

// This file contains the interface to Thor for C++ programs.
// Any of the routines described in here can raise the exceptions
// "abort" or "failure". See <some other documentation> for more
// details.

// The following definitions are used by inline routines defined here.
#include "thor_support.h"

class th_any;
class th_string;

// Startup and shutdown

extern bool th_init();
// effects     Connect to Thor.
//             This routine must be called before any other Thor routine.
//             Returns TRUE if the connection attempt succeeded

extern th_directory th_get_root();
// effects     Return the root directory object.

extern void th_shutdown();
// effects     Shuts down the session with Thor. No Thor routines
//             should be called after this call. Raises exception
//             "shutdown_failed" if it does not succeed.

// Transactions

extern void th_commit();
// checks      There should be no pending exceptions except for abort.
// effects     Attempt to commit the current transaction. If the
//             transaction cannot be committed, the "abort" exception
//             is raised. In either case, a new transaction is started.

extern void th_abort();
// effects     Aborts the current transaction, clears any pending
//             exception, and starts a new transaction.

// Exception handling support

extern bool th_catch(char const* name);
// effects     If exception named by "name" is pending, then
//             clear it and return TRUE. Else return FALSE.


extern bool th_catch_any(char const*& name);
```

```
// modifies    "name"
// effects     If there is any pending exception, store its name in "name",
//             clear the exception, and return TRUE. Else return FALSE.
//             Any returned name is guaranteed to remain valid only until
//             the next call to any Thor routine/method.

extern void th_raise(char const* name);
// XX Is this what we want
// requires    No unhandled signal exists (since a new signal cannot be
//             raised until all the previous ones have been handled)
// effects     Raises the exception given by name (the effect is as if
//             "name" was raised by Thor)

extern void th_assert_no_exceptions();
// effects     If there is any pending exception other than an "abort",
//             print an error message and kill the program.

extern void th_assert_no_abort();
// effects     If there is any pending abort?exception
//             print an error message and kill the program.

// Type conversions

th_any th_int_to_any(int i);
th_any th_bool_to_any(bool b);
th_any th_char_to_any(char c);
th_any th_null_to_any(null n);
// effects     Convert the specified primitive value to an "any" object.

// All of the following type conversion routines are "type-safe".
// They check the actual type and only perform the conversion if
// it is allowed by Theta type-checking rules. In the case where
// the conversion is disallowed, the routines raise the exception
// "wrong_type" and return an invalid object of the correct type.
// For most types, any method call on an "invalid" object kills
// the program. Special invalid objects are returned for
// primitive types:
//          int        0
//          char       0
//          bool       FALSE
//          null       nil
//          char*      ""

// The following routine is actually defined in "thor_support.h"
// extern bool th_is_invalid(th_any x);
// effects     Returns TRUE iff "x" is the special invalid object
//             returned when a Thor call raises an exception.

// The following routine is actually a macro defined in "thor_support.h"

// extern th_<foo>* th_force(th_any x, th_<foo>);
// effects     If the real type of "x" is a sub-type of "th_<foo>",
//             return "x" with type "th_<foo>".
//             Else raise the exception "wrong_type" and return an
```

```
//              invalid object of type "th_<foo>".

extern int   th_force_to_int(th_any x);
extern char  th_force_to_char(th_any x);
extern bool  th_force_to_bool(th_any x);
extern null  th_force_to_null(th_any x);
// effects      If "x" is really of the required type, then return the
//              actual value of the appropriate type.
//              Else raise the exception "wrong_type" and return an
//              invalid object of the appropriate type.

extern char* th_string_to_chars(th_string s);
// effects      Return the contents of the string "s" as a null-terminated
//              array of characters.  The contents of the returned array
//              are guaranteed to remain valid only until the next call
//              to any Thor routine.

extern th_string th_chars_to_string(char const* s);
// effects      Returns a Thor string with a copy of the contents of
//              the null-terminated string "s".

// Miscellaneous

// extern bool th_equal(th_any a1, th_any a2);
// effects      Returns TRUE iff "a1" and "a2" name the same object.

#define th_disable_futures disable_futures
// effects      Do not use futures (to facilitate debugging).

// The following environment variables can be used to gain fine-grained
// control over the execution of the underlying front-end.  Most people
// should not need to use these environment variables.
//    FE_PROGRAM     File name of fe executable
//    FE_FLAGS       Flags passed to the FE
//    FE_LOCATION    Location of already existing fe.
//                   If shared memory is being used, then this
//                   variable should be set to the shared memory key
//                   of the front-end.  Otherwise, the format should
//                   be <hostname>:<number>

#endif /* _THOR_H */
```

Bibliography

[1] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21, Washington, DC, May 1993.

[2] M. Day, B. Liskov, U. Maheshwari, and A. Myers. References to Remote Mobile Objects in Thor. *ACM Letters on Programming Languages and Systems*, March 1994.

[3] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A.C. Myers, and L. Shrira. Safe and Efficient Sharing of Persistent Objects in Thor. **To appear.**, November 1995.

[4] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A.C. Myers. Theta Reference Manual. Programming Methodology Group Memo 88, Laboratory for Computer Science, MIT, February 1995.

[5] B. Liskov, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. The Language-Independent Interface of the Thor Persistent Object System. In O. Bukhres and A. Elmagarmid, editors, *Object-Oriented Multidatabase Systems*. Prentice-Hall, 1994. Also available as Programming Methodology Group Memo 80, MIT Lab. for Computer Science, Cambridge, MA, March 1994.

[6] B. Liskov, R. Gruber, P. Johnson, and L. Shrira. A Highly Available Object Repository for Use in a Heterogeneous Distributed System. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 255–266, Martha's Vineyard, MA, September 1990. Proceedings published as *Implementing Persistent Object Bases: Principles and Practice*, A. Dearle, G. Shaw, and S. Zdonik, editors, Morgan Kaufmann, 1991. Also available as Programming Methodology Group Memo 70, MIT Lab. for Computer Science, Cambridge, MA, August 1990.

[7] B. Liskov and J. Wing. Family Values: A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[8] Barbara Liskov, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1984. Also published as Lecture Notes in Computer Science 114, G. Goos and J. Hartmanis, Eds., Springer-Verlag, 1981.

[9] U. Maheshwari and B. Liskov. Fault-Tolerant Distributed Garbage Collection in a Client-Server Object-Oriented Database. In *Proceedings of the Parallel and Distributed Information Systems*, Austin, TX, September 1994.

[10] J. Postel. Dod standard internet protocol. DARPA-Internet RFC-791, September 1981.

[11] J. Postel. Dod standard transmition control protocol. DARPA-Internet RFC-793, September 1981.