

HAFTA Checkpoint Library Architecture

Copyright ©2001-2003, Astra Network Inc. All Rights Reserved

January 3, 2003

This is documentation for a BETA version of HAFTA. Please be advised that you may encounter problems; please report them so that they may be fixed. The latest information about HAFTA is always available at <http://www.astranetwork.com/hafta/>.

Contents

1	Basic Program Structure	1
1.1	Sequences	1
1.1.1	The important features of the sequence structure	2
1.1.2	The normal function	2
1.1.3	The rollback function	3
1.1.4	The policy function	4
1.2	Checkpoints	5
1.2.1	The important features of the checkpoint structure	5
1.2.2	Details	5
2	Code Samples	6

1 Basic Program Structure

The Checkpoint Library provides a program with two levels of structure. The sequence level, and the checkpoint level.

1.1 Sequences

A *sequence* consists of an associative array style data structure containing a list of functions that the sequence will call. Each set of functions in the list, known as a *node*, contains a **normal** function, a **rollback** function, and a **policy** function.

A sequence is invoked similarly to the way one might invoke a single function, except that it is done through the API of the checkpoint library.

1.1.1 The important features of the sequence structure

The progression through the nodes in the sequence is linear or branching. It is not a dependancy map.

The nodes are anonymous and uniform. There is no need to differentiate between setup, running, or teardown functions.

The nodes are self-directing. The execution sequence is indicated to the checkpoint library by the `policy` function, which is described below.

Sequences are independant and nestable. Inter-sequence dependancies are left to the domain of the developer's code.

When the system transitions from a node, the node will be pushed onto the stack. This allows a node's policy function to roll back to a previous node should it determine that an error can't be corrected in the context of the current node.

Errors encountered during the execution of the sequence are reported by the developer's code to the checkpoint library. The checkpoint library then acts upon those errors by consulting the policy function.

1.1.2 The normal function

Each node in the sequence must have a `normal` function. This function contains the code that is called when the previous node requests the invocation of that node.

The `normal` function is called with 2 arguments:

- a sequence object
- a user data pointer

The user data pointer is a value passed to the sequence when it is invoked. This argument is the same across all function calls in the sequence. This pointer has no specific meaning to the checkpoint library, it should be used to point to a data area for the sequence to use during its execution.

The sequence object is used by the checkpoint library calls in the developer's code to determine which sequence the calls are being made from. The developer should not modify or use any data in this structure. The object should be merely passed around to the checkpoint library functions.

The `normal` function can exit in one of 3 ways.

- The first option is to simply do
`return HC_NormalSuccess(sequence, long policy_data)`. The value `policy_data` is then presented to the `policy` function, along with an indication that the `normal` function exited successfully. In the default

`policy` function, the value `policy_data` is treated as a reference to which node should be executed next.

- The second option is to do
`return HC_NormalFail(sequence, long policy_data)`. The value of `policy_data` is presented to the `policy` function, along with an indication that the `normal` function exited unsuccessfully. The default `policy` function will ignore the value of `policy_data` and request that the node be rolled back and retried up to 5 times, after which it will request a rollback to the previous node.
- The third option is to call
`HC_Panic(sequence, char *format, ...)`. Calling this function will log the event, and terminate the execution of the entire program. For the most part, you probably don't want to call this function.

1.1.3 The rollback function

Each node in the sequence that modifies the program state must have a `rollback` function. If the node does not modify the program state, this function can be excluded.

The `rollback` function is called when the `policy` function requests the node be rolled back, either to be retried, or to continue rolling back to the previous node.

The `rollback` function is called with 3 arguments:

- a sequence object
- a user data pointer
- the checkpoint value

The checkpoint value is maintained by the `normal` function and the `rollback` function. It is designed to keep track of which resources have been successfully allocated. It is further described in the Section 1.2 of this document.

When the `rollback` function is called, it is expected to undo the changes the `normal` function made to the program state.

The `rollback` function can exit in one of 3 ways.

- The first option is to simply do `return HC_RollBackSuccess(sequence, long policy_data)`. The value `policy_data` is then presented to the `policy` function, along with an indication that the `rollback` function exited successfully. In the default `policy` function, the value of `policy_data` is ignored.
- The second option is to do `return HC_RollBackFail(sequence, long policy_data)`. The value of `policy_data` is presented to the `policy` function along with an indication that the `rollback` function exited unsuccessfully. The default `policy` function will ignore the value of `policy_data` and request the rollback be retried up to 5 times, after which it will call `HC_Panic()` to terminate the process.

- The third option is to call `HC_Panic(sequence, char *format, ...)`. Calling this function will log the event, and terminate the execution of the entire program. This is, as the name suggests, a panic function, and should not be used lightly.

1.1.4 The policy function

Each node in the sequence has a `policy` function. If no `policy` function is provided, a default `policy` function implemented by the checkpoint library will be used.

The `policy` function is called in 4 cases:

- When the `normal` function exits successfully or unsuccessfully
- When the `rollback` function exits successfully or unsuccessfully

The developer's `policy` function can handle all, any, or none of these conditions. The developer's `policy` function must be designed in such a way as to fall through to the default `policy` function on unhandled or unknown event types.

When called, the `policy` function is expected to do one of 3 things:

- Request the execution of a node (including the current node)
- Request the rollback of the current node
- Request the rollback of the previous node

The `policy` function must not do anything except direct the flow of control of the sequence.

The `policy` function is called with 4 arguments:

- a sequence object
- a user data pointer
- an event - an enumeration of the event that required the invocation of this function
- `policy_data` - The value returned with that event

The user data pointer and sequence object parameters are identical to those in the `normal` and `rollback` functions.

The event parameter is a value indicating what prompted the invocation of the `policy` function - The `rollback` function failing, or the `normal` function exiting successfully, for example.

The `policy_data` argument is the value passed by the call that prompted the invocation of the `policy` function. For example, doing `return HC_NormalFail(sequence, 5)` will present '5' in the `policy_data` argument.

The default `policy` function, when called indicating the `normal` function has exited successfully, will make a call to `return HC_Normal(sequence, policy_data)`, which will invoke the node in the sequence indicated by the `policy_data` provided by the just exited `normal` function.

When the default `policy` function is invoked indicating that the `normal` or `rollback` functions exited unsuccessfully, it will do `return HC_RollbackCurrent(sequence)` up to 5 times to request that the current node be rolled back and retried.

After the 5th failure in the `normal` function, the default `policy` function will do `return HC_RollbackPrev(sequence)`, which will invoke the rollback function of this node, and then proceed to roll back the previously completed node.

After the 5th failure in the `rollback` function, the default `policy` function will call `HC_Panic()` to terminate the program.

If the current node has no `rollback` function, the default `policy` function will not attempt to roll back or retry the node, and will immediately default to rolling back the previous node.

When the default `policy` function is invoked indicating that the next node has requested that this node be rolled back, it will proceed the same as if the `normal` function had exited unsuccessfully.

1.2 Checkpoints

Checkpoints are a facility for allowing the rollback of a partially completed function. Checkpoints are a strictly linear enumeration of the resources allocated by the programmer's code.

The value of the last checkpoint set by a particular `normal` or `rollback` function is stored, then presented to the `rollback` function so that the programmer's rollback code can know what resources it needs to free and/or deconfigure.

1.2.1 The important features of the checkpoint structure

The progression through the checkpoints is strictly linear, and as such is 100% predictable. Branching is handled in the scope of the sequence structure, not the checkpoints.

The checkpoint system is designed to be inobtrusive and lightweight. It does not directly affect the flow of execution.

1.2.2 Details

During the course of the operation of the `normal` function, a resource or a number of resources may need to be allocated. These allocations are marked, by the developer's code, upon successful allocation, by a checkpoint library `HC_Checkpoint()` call.

If, for any reason, the `normal` function has an error during its operation, the checkpoint library will call the `rollback` function, if one is available, and provide it with the last value that was passed to the checkpoint library `HC_Checkpoint()`

call. Using this value, the **rollback** function will be able to determine which resources need to be deconfigured or deallocated to restore the program and sequence state to what it was before the **normal** function was called.

The gravity of this design is to structure the **rollback** function as a single switch statement containing a fall-through reverse-order list of checkpoint value cases.

2 Code Samples

Please see the Getting Started with HAFTA document and the accompanying example source code for working examples.