NAME

caesar_stack_1 – the ‘‘stack_1’’ library of OPEN/CAESAR

PURPOSE

The ‘‘stack_1’’ library provides primitives for managing a stack when performing depth-first search in the state graph.

USAGE

The ‘‘stack_1’’ library consists of:

- a predefined header file **`caesar_stack_1.h`**;
- the precompiled library file **`libcaesar.a`**, which implements the features described in **`caesar_stack_1.h`**.

Note: The ‘‘stack_1’’ library is a software layer built above the primitives offered by the ‘‘standard’’ and ‘‘edge’’ libraries, and by the *OPEN/CAESAR* graph module.

Note: The ‘‘stack_1’’ library relies on the ‘‘edge’’ library. Therefore, when using the ‘‘stack_1’’ library, there are restrictions concerning the use of the ‘‘edge’’ library primitives. These restrictions are listed in the sequel.

DESCRIPTION

Each item in the stack is basically a tuple with 3 fields:

- (1) a ‘‘label’’ field containing a label,
- (2) a ‘‘state’’ field containing a state,
- (3) an ‘‘edge’’ field containing a list of edges (see the ‘‘edge’’ library).

There is no constraint on the contents of these fields. Yet, if the stack is used for a depth-first search in the state graph (see below) it is likely that the following invariants hold:

- the state field of the stack base is the initial state of the graph;
- the label field of the stack base is undefined;
- the state field of the stack top is the current state;
- the label and state fields of the stack determine the path leading from the initial state to the current state. If, for a given stack item (different from the top), the state field is equal to S_1, and if, for the immediately above stack item, the label and state fields are respectively equal to L and S_2, then ‘‘(S_1, L, S_2)’’ is an edge of the graph;
- if, for a given stack item, the state field is equal to S, then the ‘‘edge’’ field of this item contains a list of edges outgoing from state S; more precisely, it is the list of edges that have not been explored yet.
- the lists of edges associated to the stack items are pairwise disjoint. Said differently, the edge fields respectively attached to different stack items do not have shared items in common.

FEATURES

....

`CAESAR_TYPE_STACK_1`

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_STACK_1;
```

This type denotes a pointer to the concrete representation of a stack. The stack representation is supposed to be “opaque”.

....

CAESAR_TYPE_OVERFLOW_FUNCTION_STACK_1

```
typedef void (*CAESAR_TYPE_OVERFLOW_FUNCTION_STACK_1) (CAESAR_TYPE_STACK_1);
```

CAESAR_TYPE_OVERFLOW_FUNCTION_STACK_1 is the “pointer to an overflow procedure” type used in the “stack_1” library. An overflow procedure takes one parameter of type **CAESAR_TYPE_STACK_1**. Examples of overflow procedures are **CAESAR_OVERFLOW_SIGNAL_STACK_1()**, **CAESAR_OVERFLOW_ABORT_STACK_1()**, and **CAESAR_OVERFLOW_IGNORE_STACK_1()** defined below.

....

CAESAR_OVERFLOW_SIGNAL_STACK_1

```
void CAESAR_OVERFLOW_SIGNAL_STACK_1 (CAESAR_K)
    CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure is a possible action that can be performed in case the stack pointed to by **CAESAR_K** overflows (because there is not enough memory to store new items).

It first prints an error message to the standard output, and also various statistics about the stack (including the number of items that could not be stored in memory). Then, it returns. Practically, this means that some portions of the graph will not be explored, but an error message will be issued.

....

CAESAR_OVERFLOW_ABORT_STACK_1

```
void CAESAR_OVERFLOW_ABORT_STACK_1 (CAESAR_K)
    CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure is a possible action that can be performed in case the stack pointed to by **CAESAR_K** overflows (because there is not enough memory to store new items).

It first prints an error message to the standard output, and also various statistics about the stack (including the number of items that could not be stored in memory). Then, it aborts the program using the C function **exit(3)**. The error code 1 is returned.

....

CAESAR_OVERFLOW_IGNORE_STACK_1

```
void CAESAR_OVERFLOW_IGNORE_STACK_1 (CAESAR_K)
    CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure is a possible action that can be performed in case the stack pointed to by **CAESAR_K** overflows (because there is not enough memory to store new items).

This procedure does nothing and returns. Practically, this means that some portions of the graph will not be explored; they are silently ignored.

....

CAESAR_INIT_STACK_1

```
void CAESAR_INIT_STACK_1 ()
   { ... }
```

This initialization procedure must be called before using any other primitive of the ‘‘stack_1’’ library.

This procedure calls internally the initialization procedure of the ‘‘edge’’ library; the call is done as follows:

```
CAESAR_INIT_EDGE (0, 1, 1, CAESAR_SIZE_POINTER(), CAESAR_ALIGNMENT_POINTER());
```

Consequently, when using the ‘‘stack_1’’ library, it is forbidden:

- to call **CAESAR_INIT_EDGE()** directly (which would result in several calls to this procedure with undefined results);
- to use any primitive of the ‘‘edge’’ library relying on the existence of the ‘‘previous state’’ field.

....

CAESAR_CREATE_STACK_1

```
void CAESAR_CREATE_STACK_1 (CAESAR_K, CAESAR_ORDER, CAESAR_OVERFLOW)
   CAESAR_TYPE_STACK_1 *CAESAR_K;
   CAESAR_TYPE_NATURAL CAESAR_ORDER;
   CAESAR_TYPE_OVERFLOW_FUNCTION_STACK_1 CAESAR_OVERFLOW;
   { ... }
```

This procedure allocates a stack using **CAESAR_CREATE()** and assigns its address to ***CAESAR_K**. If the allocation fails, the **NULL** value is assigned to ***CAESAR_K**.

Note: because **CAESAR_TYPE_STACK_1** is a pointer type, any variable **CAESAR_K** of type **CAESAR_TYPE_STACK_1** must be allocated before used, for instance using:

```
CAESAR_CREATE_STACK_1 (&CAESAR_K, ...);
```

The actual value of the formal parameter **CAESAR_ORDER** will be stored and associated to the stack pointed to by ***CAESAR_K**. This parameter follows the same conventions as the formal parameter **CAESAR_ORDER** of the **CAESAR_CREATE_EDGE_LIST()** procedure of the ‘‘edge’’ library. It will be used subsequently to determine the order of the list of edges contained in the ‘‘edge’’ fields of the items of the stack pointed to by ***CAESAR_K**. See below for more details.

The actual value of the formal parameter **CAESAR_OVERFLOW** will be stored and associated to the stack pointed to by ***CAESAR_K**. It will be used subsequently to determine the action to take if the stack pointed to by ***CAESAR_K** overflows: in this case, the procedure pointed to by **CAESAR_OVERFLOW** will be called with the overflowing stack ***CAESAR_K** passed as actual parameter.

The above procedures **CAESAR_OVERFLOW_SIGNAL_STACK_1()**,

CAESAR_OVERFLOW_ABORT_STACK_1 (), and **CAESAR_OVERFLOW_IGNORE_STACK_1 ()**, can be used as actual values for the formal parameter **CAESAR_OVERFLOW**.

If the actual value of the formal parameter **CAESAR_OVERFLOW** is **NULL**, it is replaced by the default value **CAESAR_OVERFLOW_SIGNAL_STACK_1**.

....

CAESAR_DELETE_STACK_1

```
void CAESAR_DELETE_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 *CAESAR_K;
   { ... }
```

This procedure frees the memory space corresponding to the stack pointed to by ***CAESAR_K** using **CAESAR_DELETE ()**. Each stack item is also freed, as well as each item of the ‘‘edge’’ field of each stack item. Afterwards, the **NULL** value is assigned to ***CAESAR_K**.

....

CAESAR_PURGE_STACK_1

```
void CAESAR_PURGE_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure empties the stack pointed to by **CAESAR_K** without deleting it. Each stack item is freed, as well as each item of the ‘‘edge’’ field of each stack item. Afterwards, the stack is exactly in the same state as after its creation using **CAESAR_CREATE_STACK_1 ()**.

....

CAESAR_COPY_STACK_1

```
void CAESAR_COPY_STACK_1 (CAESAR_K1, CAESAR_K2, CAESAR_FULL)
   CAESAR_TYPE_STACK_1 CAESAR_K1;
   CAESAR_TYPE_STACK_1 CAESAR_K2;
   CAESAR_TYPE_BOOLEAN CAESAR_FULL;
   { ... }
```

This procedure empties the stack pointed to by **CAESAR_K1** using **CAESAR_PURGE_STACK_1 ()**. This stack must have been created previously using **CAESAR_CREATE_STACK_1 ()**.

Afterwards, the contents of the stack pointed to by **CAESAR_K2** are copied to the stack pointed to by **CAESAR_K1**. For each item of the stack pointed to by **CAESAR_K2**, a duplicated item is allocated and inserted into the stack pointed to by **CAESAR_K1**. Said differently, after the copy, both stacks do not have shared items in common.

If **CAESAR_FULL** is equal to zero, the ‘‘edge’’ fields of all items in the stack pointed to by **CAESAR_K1** are set to NULL; the ‘‘edge’’ fields of the items in the stack pointed to by **CAESAR_K2** are not duplicated. This is useful for storing a path leading from the initial state to the current state.

In case of memory shortage, the overflow procedure associated with **CAESAR_K1** is called with the actual parameter **CAESAR_K1**.

....

CAESAR_DEPTH_STACK_1

```
CAESAR_TYPE_NATURAL CAESAR_DEPTH_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This function returns the number of items in the stack pointed to by **CAESAR_K**. It returns 0 if this stack is empty.

Note: the depth of a stack is the number of states (not the number of labels) stored in the stack. Even if a stack contains only a single state (in a depth-first search, this state is likely to be the initial state of the graph), the depth of the stack will be 1, not 0.

....

CAESAR_BREADTH_STACK_1

```
CAESAR_TYPE_NATURAL CAESAR_BREADTH_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This function returns the number of items that have not been explored yet in the stack pointed to by **CAESAR_K**. More precisely, it returns the sum, for all stack items, of the respective lengths of the ‘‘edge’’ fields of these items.

....

CAESAR_TOP_STATE_STACK_1

```
CAESAR_TYPE_STATE CAESAR_TOP_STATE_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This function returns a pointer to the ‘‘state’’ field of the item on the top of the stack pointed to by **CAESAR_K**. If the stack is empty, the result is undefined.

....

CAESAR_TOP_LABEL_STACK_1

```
CAESAR_TYPE_LABEL CAESAR_TOP_LABEL_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This function returns a pointer to the ‘‘label’’ field of the item on the top of the stack pointed to by **CAESAR_K**. If the stack is empty, the result is undefined.

....

CAESAR_TOP_EDGE_STACK_1

```
CAESAR_TYPE_EDGE CAESAR_TOP_EDGE_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
```

```
      { ... }
```

This function returns a pointer to the ‘‘edge’’ field of the item on the top of the stack pointed to by **CAESAR_K**. If the stack is empty, the result is undefined.

....

CAESAR_EMPTY_STACK_1

```
   CAESAR_TYPE_BOOLEAN CAESAR_EMPTY_STACK_1 (CAESAR_K)
      CAESAR_TYPE_STACK_1 CAESAR_K;
      { ... }
```

This function returns a value different from 0 if the stack pointed to by **CAESAR_K** is empty, and 0 otherwise. **CAESAR_EMPTY_STACK_1 (CAESAR_K)** is always equivalent to:

```
CAESAR_DEPTH_STACK_1 (CAESAR_K) == 0
```

....

CAESAR_EXPLORED_STACK_1

```
   CAESAR_TYPE_BOOLEAN CAESAR_EXPLORED_STACK_1 (CAESAR_K)
      CAESAR_TYPE_STACK_1 CAESAR_K;
      { ... }
```

This function returns a value different from 0 if the ‘‘edge’’ field of the item on the top of the stack pointed to by **CAESAR_K** is equal to **NULL** (i.e., the empty edge list), and 0 otherwise. If the stack is empty, the result is undefined. **CAESAR_EXPLORED_STACK_1 (CAESAR_K)** is always equivalent to:

```
*(CAESAR_TOP_EDGE_STACK_1 (CAESAR_K)) == NULL
```

....

CAESAR_CREATE_TOP_EDGE_STACK_1

```
   void CAESAR_CREATE_TOP_EDGE_STACK_1 (CAESAR_K)
      CAESAR_TYPE_STACK_1 CAESAR_K;
      { ... }
```

This procedure computes the list of the edges going out from the ‘‘state’’ field of the top of the stack pointed to by **CAESAR_K**, and assigns the result to the ‘‘edge’’ field of the stack top.

If the stack is empty, or if the ‘‘edge’’ field of the stack top is not equal to the empty list when the procedure is called, the result is undefined.

This is done by calling the **CAESAR_CREATE_EDGE_LIST()** procedure of the ‘‘edge’’ library. The actual value given to the formal parameter **CAESAR_ORDER** of this procedure is equal to the actual value of the formal parameter **CAESAR_ORDER** at the time the stack was created using **CAESAR_CREATE_STACK_1()**.

In case of memory shortage, either when allocating the new item or the list of its outgoing edges, the overflow procedure associated with **CAESAR_K** is called with the actual parameter **CAESAR_K**.

The functions **CAESAR_CREATION_EDGE_LIST()** and **CAESAR_TRUNCATION_EDGE_LIST()** can be used in the overflow procedure. They can also be used after any call to **CAESAR_CREATE_TOP_EDGE_STACK_1()**, assuming that the overflow procedure has not aborted the program.

....

CAESAR_DELETE_TOP_EDGE_STACK_1

```
void CAESAR_DELETE_TOP_EDGE_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure frees the list of the edges in the ‘‘edge’’ field of the top of the stack pointed to by **CAESAR_K**. Afterwards, the **NULL** value is assigned to the ‘‘edge’’ field of the stack top.

If the stack is empty, the result is undefined.

....

CAESAR_PUSH_STACK_1

```
void CAESAR_PUSH_STACK_1 (CAESAR_K, CAESAR_L, CAESAR_S)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   CAESAR_TYPE_LABEL CAESAR_L;
   CAESAR_TYPE_STATE CAESAR_S;
   { ... }
```

This procedure allocates a new item, using **CAESAR_CREATE()**, and pushes it onto the top of the stack pointed to by **CAESAR_K**.

The label pointed to by **CAESAR_L** is copied into the ‘‘label’’ field of the new stack top. However, if **CAESAR_L** is equal to **NULL**, the ‘‘label’’ field of the new stack top is left undefined (this is useful for pushing the base when the stack is still empty).

The state pointed to by **CAESAR_S** is copied into the ‘‘state’’ field of the new stack top. However, if **CAESAR_S** is equal to **NULL**, the ‘‘state’’ field of the new stack top is left undefined (this is useful for pushing the base when the stack is still empty).

The ‘‘edge’’ field of the new stack top is initialized to **NULL**.

In case of memory shortage when allocating the new item, the overflow procedure associated with **CAESAR_K** is called with the actual parameter **CAESAR_K**.

....

CAESAR_POP_STACK_1

```
void CAESAR_POP_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure pops the item on the top of the stack pointed to by **CAESAR_K**. This item is freed using

CAESAR_DELETE().

If the stack is empty, or if the ‘‘edge’’ field of the old stack top is not equal to the empty list, the result is undefined.

....

CAESAR_SWAP_STACK_1

```
void CAESAR_SWAP_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure removes the first item of the list of edges pointed to by the ‘‘edge’’ field of the top of the stack pointed to by **CAESAR_K**, and pushes it onto the top of the stack.

If the stack is empty, or if the ‘‘edge’’ field of the old stack top is equal to the empty list, the result is undefined.

....

CAESAR_REJECT_STACK_1

```
void CAESAR_REJECT_STACK_1 (CAESAR_K)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure removes the first item of the list of edges pointed to by the ‘‘edge’’ field of the top of the stack pointed to by **CAESAR_K**. This item is freed using **CAESAR_DELETE()**.

If the stack is empty, or if the ‘‘edge’’ field of the old stack top is equal to the empty list, the result is undefined.

....

CAESAR_FORMAT_STACK_1

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_STACK_1 (CAESAR_K, CAESAR_FORMAT)
   CAESAR_TYPE_STACK_1 CAESAR_K;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the stack pointed to by **CAESAR_K** will be printed by the procedure **CAESAR_PRINT_STACK_1()** (see below). Currently, the following formats are available:

- With format 0, statistical information about the stack is displayed such as: current depth, memory size, etc.
- With format 1, the items are printed from the stack base to the stack top. For each item, the ‘‘label’’ field is printed; however, the ‘‘label’’ field of the stack base is not printed, since it is supposed to be undefined. The ‘‘state’’ and ‘‘edge’’ fields are not printed.

  This format can be used to display the execution sequence leading from the initial state to the current state.

- With format 2, the items are printed from the stack base to the stack top. For each item, the ‘‘label’’ and ‘‘state’’ fields are printed; however, the ‘‘label’’ field of the stack base is not printed, since it is supposed to be undefined. The ‘‘edge’’ fields are not printed.

  This format is intended mainly for debugging purpose.
- With format 3, the items are printed from the stack top to the stack base. For each item, the ‘‘label’’ and ‘‘state’’ fields are printed; however, the ‘‘label’’ field of the stack base is not printed, since it is supposed to be undefined. The ‘‘edge’’ fields are not printed.

  This format is intended mainly for debugging purpose.
- With format 4, the items are printed from the stack base to the stack top. For each item, the ‘‘label’’, ‘‘state’’, and ‘‘edge’’ fields are printed; however, the ‘‘label’’ field of the stack base is not printed, since it is supposed to be undefined.

  This format is intended mainly for debugging purpose.
- With format 5, the items are printed from the stack top to the stack base. For each item, the ‘‘label’’, ‘‘state’’, and ‘‘edge’’ fields are printed; however, the ‘‘label’’ field of the stack base is not printed, since it is supposed to be undefined.

  This format is intended mainly for debugging purpose.
- (no other format available yet).

Note: whatever the format chosen, the stack will be displayed in a form compatible with the SEQ format defined in the **seq** manual page of CADP.

By default, the current format of each stack is initialized to 0.

When called with **CAESAR_FORMAT** between 0 and 5, this function sets the current format of **CAESAR_K** to **CAESAR_FORMAT** and returns an undefined result.

When called with another value of **CAESAR_FORMAT**, this function does not modify the current format of **CAESAR_K** but returns a result defined as follows. If **CAESAR_FORMAT** is equal to the constant **CAESAR_CURRENT_FORMAT**, the result is the value of the current format of **CAESAR_K**. If **CAESAR_FORMAT** is equal to the constant **CAESAR_MAXIMAL_FORMAT**, the result is the maximal format value (i.e., 5). In all other cases, the effect of this function is undefined.

....

CAESAR_MAX_FORMAT_STACK_1

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_STACK_1 ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as it might be removed from future versions of the *OPEN/CAESAR*. Use function **CAESAR_FORMAT_STACK_1 ()** instead, called with argument **CAESAR_MAXIMAL_FORMAT**.

This function returns the maximal format value available for printing stacks.

....

CAESAR_PRINT_STACK_1

```
void CAESAR_PRINT_STACK_1 (CAESAR_FILE, CAESAR_K)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure prints to file **CAESAR_FILE** a text containing information about the stack pointed to by **CAESAR_K**. The nature of the information is determined by the current format of the stack pointed to by **CAESAR_K**.

Before this procedure is called, **CAESAR_FILE** must have been properly opened, for instance using **fopen(3)**.

EXAMPLE

The following portion of C code implements a standard depth-first search using the above primitives:

```
#include "caesar_stack_1.h"

int main ()
{
CAESAR_TYPE_STACK_1 caesar_k;

CAESAR_INIT_GRAPH ();
CAESAR_INIT_STACK_1 ();

CAESAR_CREATE_STACK_1 (&caesar_k, 0, NULL);
CAESAR_PUSH_STACK_1 (caesar_k, NULL, NULL);
CAESAR_START_STATE (CAESAR_TOP_STATE_STACK_1 (caesar_k));
CAESAR_CREATE_TOP_EDGE_STACK_1 (caesar_k);

while (! CAESAR_EMPTY_STACK_1 (caesar_k)) {
        if (CAESAR_EXPLORED_STACK_1 (caesar_k))
                CAESAR_POP_STACK_1 (caesar_k);
        else if /* first successor already known */
                CAESAR_REJECT_STACK_1 (caesar_k);
        else    {
                CAESAR_SWAP_STACK_1 (caesar_k);
                CAESAR_CREATE_TOP_EDGE_STACK_1 (caesar_k);
                /* add new top in the heap */
             }
        }
exit (0);
}
```

....

AUTHOR(S)

Hubert Garavel

FILES

$CADP/incl/caesar_graph.h	interface of the graph module
$CADP/incl/caesar_*.h	interfaces of the storage module
$CADP/bin.‘arch‘/libcaesar.a	object code of the storage module
$CADP/src/open_caesar/*.c	source code of various exploration modules
$CADP/com/lotos.open	shell script to run OPEN/CAESAR

SEE ALSO

Reference Manuals of OPEN/CAESAR, CAESAR, and CAESAR.ADT, **lotos.open**(LOCAL), **caesar**(LOCAL), **caesar.adt**(LOCAL)

Additional information is available from the CADP Web page located at http://cadp.inria.fr

Directives for installation are given in files **$CADP/INSTALLATION_*.**

Recent changes and improvements to this software are reported and commented in file **$CADP/HISTORY.**

BUGS

Known bugs are described in the Reference Manual of OPEN/CAESAR. Please report new bugs to cadp@inria.fr