NAME

caesar_graph – the ‘‘graph’’ library of OPEN/CAESAR

PURPOSE

The ‘‘graph module’’ provides a C representation for the states and the labels of the transition system generated from the source program. It provides primitives to handle those states and labels.

It also provides primitives to compute the transition relation (i.e., primitives to compute the initial state and the successors of any given state).

The application programming interface specified by the graph module is language-independent: various languages can be implemented so as to comply with the ‘‘graph module’’ interface.

Note: The functions and procedures specified below should not perform input/output operations on the standard input and standard output, especially reading from the standard input or writing to the standard output. As a general principle, access to standard input and standard output is reserved to the *OPEN/CAESAR* application programs (and not to the graph module). If the functions of the graph module want to emit debugging information, this information should be sent to the standard error or, preferably, to a named, unbuffered log file. Otherwise, some *OPEN/CAESAR* application programs that rely on the standard input and standard output (e.g., the *OPEN/CAESAR* Interactive Simulator) might not function properly if the graph module performs conflicting accesses to the standard input or the standard output.

USAGE

The graph module consists of:

- A predefined header file **`caesar_graph.h`**, which can be found in the *OPEN/CAESAR* package. This file is an interface specification describing a set of C types, procedures, and functions. Conceptually, it can be seen as an abstract data type, with sorts and operations.
- A C file **`spec.c`**, which is generated from the source program (where **`spec`** can be any valid filename). For instance, in the case of a LOTOS program **`spec.lotos`**, file **`spec.c`** is generated by *CAESAR* with the **`-open`** option. The **`spec.c`** file provides an implementation for all the features described in **`caesar_graph.h`**.

The **`spec.c`** must start with the two following directives:

```
#define CAESAR_GRAPH_IMPLEMENTATION ...
#include "caesar_graph.h"
```

where **`...`** should be replaced by an integer number corresponding to the version of the graph module interface for which **`spec.c`** has been produced. As far as possible, this integer number will be used to preserve backward compatibility in case of future modifications of the graph module interface. Concretely, this integer number is obtained by taking the version number given for *OPEN/CAESAR* at the bottom of file **`$CADP/VERSION`**, then by multiplying by 10 this version number (considered as real number) in order to turn it into an integer number (since the C preprocessor only handles integers). For instance, if the version of *OPEN/CAESAR* is 2.4 in the **`$CADP/VERSION`** file, then **`CAESAR_GRAPH_IMPLEMENTATION`** should be set to 24.

The features defined in ‘**`caesar_graph.h`**’’ are described below.

Note: The graph module uses the primitives offered by the ‘‘standard’’ and ‘‘version’’ libraries.

GENERAL FEATURES

.....

CAESAR_GRAPH_COMPILER

```
CAESAR_TYPE_STRING CAESAR_GRAPH_COMPILER ()
   { ... }
```

This function returns a character string containing the name (in upper case letters) of the compiler tool which generated the C program **spec.c**. For instance, if **spec.c** is generated by *CAESAR* from a LOTOS program, the result of function **CAESAR_GRAPH_COMPILER()** is the character string **"CAESAR"**.

Note: It is not allowed to modify the character string returned by **CAESAR_GRAPH_COMPILER()** nor to free it, for instance using **free(3)**.

Note: The contents of the character string returned by **CAESAR_GRAPH_COMPILER()** may be destroyed by a subsequent call to this function.

.....

CAESAR_GRAPH_VERSION

```
CAESAR_TYPE_VERSION CAESAR_GRAPH_VERSION ()
   { ... }
```

This function returns the version number of the compiler tool which generated the C program **spec.c**.

.....

CAESAR_INIT_GRAPH

```
void CAESAR_INIT_GRAPH ()
   { ... }
```

This procedure contains initialization actions for the graph module. It must be invoked once before using any routine of this module.

Implementation note: This procedure should invoke the **CAESAR_CHECK_VERSION()** procedure (see the corresponding description in the ''version'' library) in order to detect version clashes between **spec.c** and **caesar_graph.h**.

STATE FEATURES

.....

CAESAR_BODY_STATE

```
typedef struct CAESAR_STRUCT_STATE { ... } CAESAR_BODY_STATE;
```

This type denotes the actual implementation of states in the labelled transition system. Each state is basically a structure named **CAESAR_STRUCT_STATE**. Thus, each state can be seen as a byte string of fixed size (see function **CAESAR_SIZE_STATE()** below) with definite alignment constraints (see function **CAESAR_ALIGNMENT_STATE()** below), and all the states have the same size.

State definition is ''opaque'': the detailed definition of **CAESAR_STRUCT_STATE** and **CAESAR_BODY_STATE** is only available in include mode, but not in link mode. Therefore, making assumptions about the fields of structure **CAESAR_STRUCT_STATE** is not advisable.

.....

CAESAR_TYPE_STATE

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_STATE;
```

This type denotes a pointer to the (opaque) representation of states. It is given an abstract definition in file **caesar_graph.h** and should be redefined in **spec.c**.

Concretely, **CAESAR_TYPE_STATE** should be defined as a pointer to a structure named **CAESAR_STRUCT_STATE** or, equivalently, a pointer to type **CAESAR_BODY_STATE** (see above).

.....

CAESAR_SIZE_STATE

```
CAESAR_TYPE_NATURAL CAESAR_SIZE_STATE ()
   { ... }
```

This function returns the state size (in bytes), which is always greater than 0.

Implementation note: Practically, in **caesar_graph.h**, function **CAESAR_SIZE_STATE()** is defined as follows:

```
#define CAESAR_SIZE_STATE() CAESAR_HINT_SIZE_STATE
```

where **CAESAR_HINT_SIZE_STATE** is a variable defined by:

```
extern CAESAR_TYPE_NATURAL CAESAR_HINT_SIZE_STATE;
```

This variable should be defined, properly initialized, and exported by **spec.c**. It should neither be used nor assigned in any other program than **spec.c**. This variable is only introduced for efficiency reasons, i.e., to avoid the cost of a function call.

.....

CAESAR_HASH_SIZE_STATE

```
CAESAR_TYPE_NATURAL CAESAR_HASH_SIZE_STATE ()
   { ... }
```

This function returns a number of bytes N such that, for any state pointed to by a variable **CAESAR_S**, one can compute a hash function which takes into account the value of the following bytes: **CAESAR_S** [0], **CAESAR_S** [1], ... and **CAESAR_S** [N-1].

Note: It is always true that:

0 < **CAESAR_HASH_SIZE_STATE ()** <= **CAESAR_SIZE_STATE ()**

but it is possible that:

CAESAR_HASH_SIZE_STATE () < **CAESAR_SIZE_STATE ()**

especially if the state vector contains variables of pointer types. By using this function, users can write their own hash functions.

Implementation note: Practically, in **caesar_graph.h**, function **CAESAR_HASH_SIZE_STATE()** is defined as follows:

```
#define CAESAR_HASH_SIZE_STATE() CAESAR_HINT_HASH_SIZE_STATE
```

where **CAESAR_HINT_HASH_SIZE_STATE** is a variable defined by:

```
extern CAESAR_TYPE_NATURAL CAESAR_HINT_HASH_SIZE_STATE;
```

This variable should be defined, properly initialized, and exported by **spec.c**. It should neither be used nor assigned in any other program than **spec.c**. This variable is only introduced for efficiency reasons, i.e., to avoid the cost of a function call.

.....

CAESAR_ALIGNMENT_STATE

```
CAESAR_TYPE_NATURAL CAESAR_ALIGNMENT_STATE ()
   { ... }
```

This function returns the alignment factor (in bytes) for states. The alignment factor is always a power of two, usually 1, 2, 4, or 8. Any byte string representing a state must be aligned on a boundary that is an even multiple of the alignment factor.

Implementation note: Practically, in **caesar_graph.h**, function **CAESAR_ALIGNMENT_STATE()** is defined as follows:

```
#define CAESAR_ALIGNMENT_STATE() CAESAR_HINT_ALIGNMENT_STATE
```

where **CAESAR_HINT_ALIGNMENT_STATE** is a variable defined by:

```
extern CAESAR_TYPE_NATURAL CAESAR_HINT_ALIGNMENT_STATE;
```

This variable should be defined, properly initialized, and exported by **spec.c**. It should neither be used nor assigned in any other program than **spec.c**. This variable is only introduced for efficiency reasons, i.e., to avoid the cost of a function call.

.....

CAESAR_CREATE_STATE

```
void CAESAR_CREATE_STATE (CAESAR_S)
   CAESAR_TYPE_STATE *CAESAR_S;
   { ... }
```

This procedure allocates a byte string of length **CAESAR_SIZE_STATE()** using **CAESAR_CREATE()** and assigns its address to ∗**CAESAR_S**. If the allocation fails, the **NULL** value is assigned to ∗**CAESAR_S**.

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

```
CAESAR_CREATE_STATE (&CAESAR_S);
```

However, it is not necessary to use **CAESAR_CREATE_STATE()** to perform the allocation. Instead, users can allocate states into their own data structures (tables, lists, ...)

Implementation note: It is not necessary to define **CAESAR_CREATE_STATE()** in **spec.c** because **caesar_graph.h** already implements this procedure using a macro-definition.

.....

CAESAR_DELETE_STATE

```
void CAESAR_DELETE_STATE (CAESAR_S)
   CAESAR_TYPE_STATE *CAESAR_S;
   { ... }
```

This procedure frees the byte string of length **CAESAR_SIZE_STATE()** pointed to by ∗**CAESAR_S** using **CAESAR_DELETE()**. Afterwards, the **NULL** value is assigned to ∗**CAESAR_S**.

Implementation note: It is not necessary to define **CAESAR_DELETE_STATE()** in **spec.c** because **caesar_graph.h** already implements this procedure using a macro-definition.

.....

CAESAR_COPY_STATE

```
void CAESAR_COPY_STATE (CAESAR_S1, CAESAR_S2)
   CAESAR_TYPE_STATE CAESAR_S1;
   CAESAR_TYPE_STATE CAESAR_S2;
   { ... }
```

This procedure copies the state pointed to by **CAESAR_S2** onto the state pointed to by **CAESAR_S1**.

Note: Parameter **CAESAR_S2** must point to a memory location allocated before procedure **CAESAR_COPY_STATE()** is invoked.

Implementation note: It is not necessary to define **CAESAR_COPY_STATE()** in **spec.c** because **caesar_graph.h** already implements this procedure using a macro-definition.

.....

CAESAR_COMPARE_STATE

```
CAESAR_TYPE_BOOLEAN CAESAR_COMPARE_STATE (CAESAR_S1, CAESAR_S2)
   CAESAR_TYPE_STATE CAESAR_S1;
   CAESAR_TYPE_STATE CAESAR_S2;
   { ... }
```

This function returns a value different from 0 if both states pointed to by **CAESAR_S1** and **CAESAR_S2** are identical, or 0 if they are not.

.....

CAESAR_HASH_STATE

```
CAESAR_TYPE_NATURAL CAESAR_HASH_STATE (CAESAR_S, CAESAR_MODULUS)
   CAESAR_TYPE_STATE CAESAR_S;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-code value for the state pointed to by **CAESAR_S** and returns this value, which must be in the range 0..**CAESAR_MODULUS**-1.

.....

CAESAR_FORMAT_STATE

```
void CAESAR_FORMAT_STATE (CAESAR_FORMAT)
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This procedure allows to control the format under which states are printed by procedures **CAESAR_PRINT_STATE_HEADER()**, **CAESAR_PRINT_STATE()**, and **CAESAR_DELTA_STATE()** (see below). Currently, the following formats are available:

- If the current state format is 0, each state is printed on a single line, which should not be terminated by a new-line character (“**\n**”).

  CAESAR-specific note: states generated from LOTOS programs are printed as a pair consisting of a marking part (marked places in the parallel components) and a context part (values of state variables).

  CAESAR-specific note: only one state format (numbered 0) is available.
- (no other format available yet).

By default, the current state format is initialized to 0.

When called with **CAESAR_FORMAT** between 0 and the maximal format value supported, this fonction sets the current state format to **CAESAR_FORMAT** and returns an undefined result.

When called with another value of **CAESAR_FORMAT**, this function does not modify the current state format 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 state format. If **CAESAR_FORMAT** is equal to the constant **CAESAR_MAXIMAL_FORMAT**, the result is the maximal format value (e.g., 0 in the case of *CAESAR*). In all other cases, the effect of this function is undefined.

.....

CAESAR_MAX_FORMAT_STATE

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_STATE ()
   { ... }
```

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_STATE()** instead, called with argument **CAESAR_MAXIMAL_FORMAT**.

This function returns the highest format available for state printing, i.e., the highest acceptable value for the parameter **CAESAR_FORMAT** of function **CAESAR_FORMAT_STATE()**.

.....

CAESAR_PRINT_STATE_HEADER

```
void CAESAR_PRINT_STATE_HEADER (CAESAR_FILE)
   CAESAR_TYPE_FILE CAESAR_FILE;
   { ... }
```

This procedure prints to file **CAESAR_FILE** information about the structure of states. The nature of the

information is determined by the current state format (see procedure **CAESAR_FORMAT_STATE()** above). This procedure is to be used in conjunction with the next one.

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

.....

CAESAR_PRINT_STATE

```
void CAESAR_PRINT_STATE (CAESAR_FILE, CAESAR_S)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_STATE CAESAR_S;
   { ... }
```

This procedure prints to file **CAESAR_FILE** information about the contents of the state pointed to by **CAESAR_S**. The nature of the information is determined by the current state format (see procedure **CAESAR_FORMAT_STATE()** above).

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

.....

CAESAR_DELTA_STATE

```
void CAESAR_DELTA_STATE (CAESAR_FILE, CAESAR_S1, CAESAR_S2)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_STATE CAESAR_S1;
   CAESAR_TYPE_STATE CAESAR_S2;
   { ... }
```

This procedure prints to file **CAESAR_FILE** information about the differences (''delta'') between both states pointed to by **CAESAR_S1** and **CAESAR_S2** respectively. The nature of the information is determined by the current state format (see procedure **CAESAR_FORMAT_STATE()** above).

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

LABEL FEATURES

.....

CAESAR_BODY_LABEL

```
typedef struct CAESAR_STRUCT_LABEL { ... } CAESAR_BODY_LABEL;
```

This type denotes the actual implementation of labels in the labelled transition system. Each label is basically a structure named **CAESAR_STRUCT_LABEL**. Thus, each label is a byte string of fixed size (see function **CAESAR_SIZE_LABEL()** below) with definite alignment constraints (see function **CAESAR_ALIGNMENT_LABEL()** below), and all the labels have the same size. In a first approach, the label representation is supposed to be ''opaque'': the detailed definition of **CAESAR_STRUCT_LABEL** and **CAESAR_BODY_LABEL** is only available in include mode, but not in link mode. Therefore, making assumptions about the fields of structure **CAESAR_STRUCT_LABEL** is not advisable.

In a refined approach, it is assumed that labels have an internal structure consisting of a *gate* (i.e., an identifier representing the name of a communication port) and a finite list of *experiment offers* (i.e., typed data parameters exchanged on the gate). This assumption is made without loss of generality: although the

names *gate* and *experiment offer* are borrowed from the LOTOS vocabulary, most formal description techniques for concurrent systems make a distinction between communication ports and the parameters sent or received on these ports.

Implementation note: If the transition system generated from the source program has only a single state and no transition, there are no labels at all. In such case, the C program **spec.c** generated by the compiler tool shall nevertheless provide an implementation for all types, functions, and procedures defined in this section and related to labels. Such an implementation will not be used (since there are no labels) but it is needed to ensure that everything compiles properly. The implementation details are left undefined: any implementation that complies with the requirements stated below, is acceptable.

.....

CAESAR_TYPE_LABEL

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_LABEL;
```

This type denotes a pointer to the (opaque) representation of labels. It is given an abstract definition in file **caesar_graph.h** and should be redefined in **spec.c**.

Concretely, **CAESAR_TYPE_LABEL** should be defined as a pointer to a structure named **CAESAR_STRUCT_LABEL** or, equivalently, a pointer to type **CAESAR_BODY_LABEL** (see above).

.....

CAESAR_SIZE_LABEL

```
CAESAR_TYPE_NATURAL CAESAR_SIZE_LABEL ()
   { ... }
```

This function returns the label size (in bytes), which is always greater than 0.

Implementation note: Practically, in **caesar_graph.h**, function **CAESAR_SIZE_LABEL()** is defined as follows:

```
#define CAESAR_SIZE_LABEL() CAESAR_HINT_SIZE_LABEL
```

where **CAESAR_HINT_SIZE_LABEL** is a variable defined by:

```
extern CAESAR_TYPE_NATURAL CAESAR_HINT_SIZE_LABEL;
```

This variable should be defined, properly initialized, and exported by **spec.c**. It should neither be used nor assigned in any other program than **spec.c**. This variable is only introduced for efficiency reasons, i.e., to avoid the cost of a function call.

.....

CAESAR_HASH_SIZE_LABEL

```
CAESAR_TYPE_NATURAL CAESAR_HASH_SIZE_LABEL ()
   { ... }
```

This function returns a number of bytes N such that, for any label pointed to by a variable **CAESAR_L**, one can compute a hash function which takes into account the value of the following bytes: **CAESAR_L** [0], **CAESAR_L** [1], ... and **CAESAR_L** [N-1].

Note: It is always true that:

0 < **CAESAR_HASH_SIZE_LABEL ()** <= **CAESAR_SIZE_LABEL ()**

but it is possible that:

CAESAR_HASH_SIZE_LABEL () < **CAESAR_SIZE_LABEL ()**

especially if the label vector contains variables of pointer types. By using this function, users can write their own hash functions.

Implementation note: Practically, in **caesar_graph.h**, function **CAESAR_HASH_SIZE_LABEL()** is defined as follows:

```
#define CAESAR_HASH_SIZE_LABEL() CAESAR_HINT_HASH_SIZE_LABEL
```

where **CAESAR_HINT_HASH_SIZE_LABEL** is a variable defined by:

```
extern CAESAR_TYPE_NATURAL CAESAR_HINT_HASH_SIZE_LABEL;
```

This variable should be defined, properly initialized, and exported by **spec.c**. It should neither be used nor assigned in any other program than **spec.c**. This variable is only introduced for efficiency reasons, i.e., to avoid the cost of a function call.

.....

CAESAR_ALIGNMENT_LABEL

```
CAESAR_TYPE_NATURAL CAESAR_ALIGNMENT_LABEL ()
   { ... }
```

This function returns the alignment factor (in bytes) for labels. The alignment factor is always a power of two, usually 1, 2, 4, or 8. Any byte string representing a label must be aligned on a boundary that is an even multiple of the alignment factor.

Implementation note: Practically, in **caesar_graph.h**, function **CAESAR_ALIGNMENT_LABEL()** is defined as follows:

```
#define CAESAR_ALIGNMENT_LABEL() CAESAR_HINT_ALIGNMENT_LABEL
```

where **CAESAR_HINT_ALIGNMENT_LABEL** is a variable defined by:

```
extern CAESAR_TYPE_NATURAL CAESAR_HINT_ALIGNMENT_LABEL;
```

This variable should be defined, properly initialized, and exported by **spec.c**. It should neither be used nor assigned in any other program than **spec.c**. This variable is only introduced for efficiency reasons, i.e., to avoid the cost of a function call.

.....

CAESAR_CREATE_LABEL

```
void CAESAR_CREATE_LABEL (CAESAR_L)
   CAESAR_TYPE_LABEL *CAESAR_L;
   { ... }
```

This procedure allocates a byte string of length **CAESAR_SIZE_LABEL()** using **CAESAR_CREATE()**

and assigns its address to ∗**CAESAR_L**. If the allocation fails, the **NULL** value is assigned to ∗**CAESAR_L**.

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

```
CAESAR_CREATE_LABEL (&CAESAR_L);
```

However, it is not necessary to use **CAESAR_CREATE_LABEL** to perform the allocation. Instead, users can allocate labels into their own data structures (tables, lists, ...)

Implementation note: It is not necessary to define **CAESAR_CREATE_LABEL()** in **spec.c** because **caesar_graph.h** already implements this procedure using a macro-definition.

.....

CAESAR_DELETE_LABEL

```
void CAESAR_DELETE_LABEL (CAESAR_L)
   CAESAR_TYPE_LABEL *CAESAR_L;
   { ... }
```

This procedure frees the byte string of length **CAESAR_SIZE_LABEL()** pointed to by ∗**CAESAR_L** using **CAESAR_DELETE()**. Afterwards, the **NULL** value is assigned to ∗**CAESAR_L**.

Implementation note: It is not necessary to define **CAESAR_DELETE_LABEL()** in **spec.c** because **caesar_graph.h** already implements this procedure using a macro-definition.

.....

CAESAR_VISIBLE_LABEL

```
CAESAR_TYPE_BOOLEAN CAESAR_VISIBLE_LABEL (CAESAR_L)
   CAESAR_TYPE_LABEL CAESAR_L;
   { ... }
```

This function returns a value different from 0 if the label pointed to by **CAESAR_L** is visible, or 0 if it is not visible (i.e., ‘‘tau’’).

.....

CAESAR_GATE_LABEL

```
CAESAR_TYPE_STRING CAESAR_GATE_LABEL (CAESAR_L)
   CAESAR_TYPE_LABEL CAESAR_L;
   { ... }
```

This function returns a pointer to a character string containing the name of the gate associated with the label pointed to by **CAESAR_L**. If this label is not visible (i.e., ‘‘tau’’), a pointer to the constant string **i** is returned.

Note: in order to determine if the gate of the label pointed to by **CAESAR_L** is not ‘‘tau’’, using:

```
CAESAR_VISIBLE_LABEL (CAESAR_L)
```

is more efficient than performing a string comparison:

```
strcmp (CAESAR_GATE_LABEL (CAESAR_L), "i") != 0
```

Note: It is not allowed to modify the character string returned by **CAESAR_GATE_LABEL()** nor to free it, for instance using **free(3)**.

Note: The contents of the character string returned by **CAESAR_GATE_LABEL()** may be destroyed by a subsequent call to this function.

.....

CAESAR_CARDINAL_LABEL

```
CAESAR_TYPE_NATURAL CAESAR_CARDINAL_LABEL (CAESAR_L)
   CAESAR_TYPE_LABEL CAESAR_L;
   { ... }
```

This function returns the number of experiment offers associated with the label pointed to by **CAESAR_L**. The gate itself does not count.

.....

CAESAR_COPY_LABEL

```
void CAESAR_COPY_LABEL (CAESAR_L1, CAESAR_L2)
   CAESAR_TYPE_LABEL CAESAR_L1;
   CAESAR_TYPE_LABEL CAESAR_L2;
   { ... }
```

This procedure copies the label pointed to by **CAESAR_L2** onto the label pointed to by **CAESAR_L1**.

Note: Parameter **CAESAR_L2** must point to a memory location allocated before procedure **CAESAR_COPY_LABEL()** is invoked.

Implementation note: It is not necessary to define **CAESAR_COPY_LABEL()** in **spec.c** because **caesar_graph.h** already implements this procedure using a macro-definition.

.....

CAESAR_COMPARE_LABEL

```
CAESAR_TYPE_BOOLEAN CAESAR_COMPARE_LABEL (CAESAR_L1, CAESAR_L2)
   CAESAR_TYPE_LABEL CAESAR_L1;
   CAESAR_TYPE_LABEL CAESAR_L2;
   { ... }
```

This function returns a value different from 0 if both labels pointed to by **CAESAR_L1** and **CAESAR_L2** are identical, or 0 if they are not.

.....

CAESAR_HASH_LABEL

```
CAESAR_TYPE_NATURAL CAESAR_HASH_LABEL (CAESAR_L, CAESAR_MODULUS)
   CAESAR_TYPE_LABEL CAESAR_L;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-code value for the label pointed to by **CAESAR_L** and returns this value, which must be in the range 0..**CAESAR_MODULUS**-1.

.....

CAESAR_PRINT_LABEL

```
void CAESAR_PRINT_LABEL (CAESAR_FILE, CAESAR_L)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_LABEL CAESAR_L;
   { ... }
```

This procedure prints to file **CAESAR_FILE** a character string describing the contents of the label pointed to by **CAESAR_L**. This string should not contain the special characters new-line (“**\n**”) or carriage-return (“**\r**”). If the label is not visible, the string printed is **i**.

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

.....

CAESAR_STRING_LABEL

```
CAESAR_TYPE_STRING CAESAR_STRING_LABEL (CAESAR_L)
   CAESAR_TYPE_LABEL CAESAR_L;
   { ... }
```

This function returns a pointer to a character string (terminated by the **'\0'** character) describing the contents of the label pointed to by **CAESAR_L**. This string should not contain the special characters new-line (“**\n**”) or carriage-return (“**\r**”). If the label is not visible, the string returned is **i**.

Note: It is not allowed to modify the character string returned by **CAESAR_STRING_LABEL()** nor to free it, for instance using **free(3)**.

Note: The contents of the character string returned by **CAESAR_STRING_LABEL()** may be destroyed by a subsequent call to this function.

.....

CAESAR_FORMAT_LABEL

```
void CAESAR_FORMAT_LABEL (CAESAR_FORMAT)
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This procedure allows to control the format under which the information attached to labels is displayed by the procedure **CAESAR_INFORMATION_LABEL()** (see below).

CAESAR-specific note: 3 different label formats (numbered from 0 to 2) are available.

By default, the current label format is initialized to 0.

When called with **CAESAR_FORMAT** between 0 and the maximal format value supported, this fonction sets the current label format to **CAESAR_FORMAT** and returns an undefined result.

When called with another value of **CAESAR_FORMAT**, this function does not modify the current label format 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 label format. If **CAESAR_FORMAT** is equal to the constant **CAESAR_MAXIMAL_FORMAT**, the result is the maximal format value (e.g., 2 in the case of *CAESAR*). In all other cases, the effect of this function is undefined.

.....

CAESAR_MAX_FORMAT_LABEL

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_LABEL ()
   { ... }
```

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_LABEL()** instead, called with argument **CAESAR_MAXIMAL_FORMAT**.

This function returns the highest format available for label printing, i.e., the highest acceptable value for the parameter **CAESAR_FORMAT** of function **CAESAR_FORMAT_LABEL()**.

.....

CAESAR_INFORMATION_LABEL

```
CAESAR_TYPE_STRING CAESAR_INFORMATION_LABEL (CAESAR_L)
   CAESAR_TYPE_LABEL CAESAR_L;
   { ... }
```

This function returns a character string containing additional information about the label pointed to by **CAESAR_L**. The nature of the information is determined by the current label format (see procedure **CAESAR_FORMAT_LABEL()** above). If the current label format is null, this function returns the empty string "".

Note: It is not allowed to modify the character string returned by **CAESAR_INFORMATION_LABEL()** nor to free it, for instance using **free(3)**.

Note: The contents of the character string returned by **CAESAR_INFORMATION_LABEL()** may be destroyed by a subsequent call to this function.

CAESAR-specific note: Currently, the following formats are available:

- If the current label format is 0: the empty string "" is returned.
- If the current label format is 1: if the label L pointed to by **CAESAR_L** is visible, the empty string "" is returned; if L is not visible and derives from some gate G hidden by a **hide** instruction, a character string containing the name of G, its definition file and definition line is returned; if L is not visible and derives from an **exit** statement, the constant character string **exit** is returned; if L is not visible and derives from an **i; ...** statement, the constant character string **i** is returned.
- If the current label format is 2: a character string containing the (unique) number of the Petri Net transition corresponding to the edge whose label is pointed to by **CAESAR_L** is returned. This transition can be found in the **.net** file generated by *CAESAR* with **-network** option.

EDGE FEATURES

.....

CAESAR_START_STATE

```
void CAESAR_START_STATE (CAESAR_S)
   CAESAR_TYPE_STATE CAESAR_S;
```

```
{ ... }
```

This procedure copies into the state pointed to by **CAESAR_S** the contents of the initial state of the labelled transition system.

Note: Parameter **CAESAR_S** must point to a memory location allocated before procedure **CAESAR_START_STATE()** is invoked.

.....

CAESAR_ITERATE_STATE

```
void CAESAR_ITERATE_STATE (CAESAR_S1, CAESAR_L, CAESAR_S2, CAESAR_LOOP)
   CAESAR_TYPE_STATE CAESAR_S1;
   CAESAR_TYPE_LABEL CAESAR_L;
   CAESAR_TYPE_STATE CAESAR_S2;
   void (*CAESAR_LOOP) (CAESAR_TYPE_STATE, CAESAR_TYPE_LABEL,
      CAESAR_TYPE_STATE);
   { ... }
```

This procedure provides an iterator which enumerates all successors of the state pointed to by **CAESAR_S1**. At each iteration, the label pointed to by **CAESAR_L** and the state pointed to by **CAESAR_S2** are assigned a new value, such that "(**CAESAR_S1**, **CAESAR_L**, **CAESAR_S2**)" is an edge of the labelled transition system. At each iteration, the C function pointed to by **CAESAR_LOOP** is invoked, with the following parameters:

```
(*CAESAR_LOOP) (CAESAR_S1, CAESAR_L, CAESAR_S2)
```

Therefore, any actual parameter supplied for the formal parameter **CAESAR_LOOP** must be a pointer to a function, say **caesar_f**, whose declaration is:

```
void caesar_f (caesar_s1, caesar_l, caesar_s2)
   CAESAR_TYPE_STATE caesar_s1;
   CAESAR_TYPE_LABEL caesar_l;
   CAESAR_TYPE_STATE caesar_s2;
   {...}
```

Note: Parameters **CAESAR_S1**, **CAESAR_L**, and **CAESAR_S2** must point to (distinct) memory locations allocated before procedure **CAESAR_ITERATE_STATE()** is invoked. In no event will **CAESAR_ITERATE_STATE()** and **CAESAR_LOOP()** allocate memory for storing **CAESAR_L** and **CAESAR_S2**.

Note: More often than not, this function will have side-effects. For instance, this function may count the number of successors, or store them in a list, a table, ...

Note: It is probably a good programming style to keep the body of this function as short as possible.

Note: The C code that implements **CAESAR_ITERATE_STATE()** in **spec.c** may be not reentrant. In such case, nested iterations will not work properly. This implies that any actual parameter **caesar_f** supplied to formal parameter **CAESAR_LOOP** must not call (directly, nor transitively) **CAESAR_ITERATE_STATE**. Practically, this restriction has no effect on breadth-first explorations. However, it affects the way of writing depth-first explorations: for a given state S, it is necessary first to compute all successors of S and to store them in some data structure, before starting to explore these successors.

CAESAR-specific note: The C code generated by *CAESAR* for **CAESAR_ITERATE_STATE()** is not reentrant (at least in the current version).

OBSOLETE FEATURES

.....

CAESAR_DUMP_LABEL

```
void CAESAR_DUMP_LABEL (CAESAR_STRING, CAESAR_L)
   CAESAR_TYPE_STRING CAESAR_STRING;
   CAESAR_TYPE_LABEL CAESAR_L;
   { ... }
```

This procedure has been removed from the ''graph module'' application programming interface in October 2002 and should no longer be implemented, nor used in *OPEN/CAESAR* application programs.

.....

CAESAR_RANK_LABEL

```
CAESAR_TYPE_NATURAL CAESAR_RANK_LABEL (CAESAR_L)
   CAESAR_TYPE_LABEL CAESAR_L;
   { ... }
```

This function has been removed from the ''graph module'' application programming interface in October 2002 and should no longer be implemented, nor used in *OPEN/CAESAR* application programs.

.....

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