THE OPEN/CÆSAR REFERENCE MANUAL

Hubert Garavel

February 13, 2020

Note: this PDF document contains hyperlinks written in grey, whereas the normal text is in black.

Contents

Abstract

This reports describes the Open/Cæsar *environment for developing specifications in* Lotos. *The* Open/Cæsar *environment is built on the* Cæsar *tool. Although* Cæsar *is mainly intended for compilation and exhaustive verification of* Lotos *specifications, the* Open/Cæsar *provides for interactive simulation, random simulation, partial verification, on-the-fly verification, sequential code generation, test generation, etc.*

In a first part, the principles and the architecture of the Open/Cæsar *environment are explained.*

In a second part, the interfaces of Open/Cæsar *modules and libraries aare presented.*

Note

The Open/Cæsar *features described in this report have reached a certain degree of maturity. However, they are subject to change in the future, in order to achieve various improvements.*

If you develop tools using the facilities provided by Open/Cæsar *— or if you plan to do so — please be sure that you have the latest version of the* Open/Cæsar *reference manual in your possession. Also, if you want to be informed and consulted about subsequent changes, please contact the author (*`Hubert.Garavel@imag.fr`*).*

Part I

Principles

Chapter 1

Objectives of the OPEN/CÆSAR project

Foreword

There exist an overview publication about the OPEN/CÆSAR project, which should be read before the present Reference Manual. The bibliographic reference is:

> Hubert Garavel. *OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing*. In Bernhard Steffen, ed., Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal), March 1998. Lecture Notes in Computer Science 1384, pp. 68–84. Full version available as INRIA Research Report RR-3352 from `http://cadp.inria.fr/publications/Garavel-92-a.html`.

1.1 Motivation

The OPEN/CÆSAR project originates from the problems reported by people using the CÆSAR tool.

Indeed, when using the "basic" CÆSAR tool (i.e., the version of CÆSAR up to and including 4.2), the *state explosion problem* often occurred: due to a lack of memory, the *labelled transition system* (called here *graph*, for short) could not be generated. Therefore, the source LOTOS specification could not be verified.

On the other hand, some existing LOTOS tools — although lacking the efficient compiling algorithms of CÆSAR — could analyze larger LOTOS specifications. Of course, full verification could not be achieved using these tools, but some debugging was allowed (by means of interactive simulation).

The problem comes from the fact that "basic" CÆSAR is "closed", in the sense that it only allows verification by exhaustive simulation (reachability analysis).

The OPEN/CÆSAR project attempts to solve this problem by "opening up" CÆSAR, so that the user can control all the parameters of the graph exploration. For instance, the user can:

- handle the states, the labels, and the edges of the graph;

- decide how many states — and which — are to be stored in memory;
- decide how many successor states of the current state — and which – are to be visited;
- choose between depth-first or breadth-first search;
- etc.

Therefore, the OPEN/CÆSAR environment goes far beyond the potentialities of "basic" CÆSAR. It still allows exhaustive simulation, but also:

- interactive simulation
- random (intensive) simulation
- "on the fly" verification
- sequential code generation
- test generation

1.2 Rationale

The OPEN/CÆSAR aims at providing a "toolkit" which should allow computer scientists to experiment quickly and efficiently new algorithms for computer-aided verification.

It should be of interest to researchers for, at least, three reasons:

1. Verification techniques must be confronted to "real-life" problems in order to assess their effectiveness. Therefore, a verification tool should be based on a widely accepted specification language, so that "real-life" specifications are available.

   ⟶ OPEN/CÆSAR operates on the LOTOS standardized by ISO. It tackles the full language, including values. Yet, dynamic creation/destruction of processes is not allowed.

2. To implement a new verification technique for a realistic specification language, it is first necessary to develop a compiler for this language. As far as standardized Formal Description Techniques ESTELLE, LOTOS, and SDL are concerned, compiler construction is a very tasking job that should be avoided if possible.

   ⟶ OPEN/CÆSAR already comes with a full LOTOS compiler, which has been continuously used and improved since 1986. It is based on a powerful technique for translating LOTOS into extended Petri nets and labelled transition systems. Also, OPEN/CÆSAR comes with an efficient abstract data type compiler, CÆSAR.ADT. Finally, it is interfaced with various verification tools, such as ALDÉBARAN and AUTO/AUTOGRAPH.

3. Modifying existing tools to implement a new verification technique is not easy. This can be even more difficult for tools which have not been specifically designed for verification purpose.

   ⟶ OPEN/CÆSAR provides a clean interface for the extended Petri nets and the labelled transition systems produced from the source LOTOS specifications. This interface has been carefully designed and it will be extended as new needs are identified.

To summarize, OPEN/CÆSAR should allow news ideas can be implemented easily and to be confronted to "real" specifications, without the task of developing a compiler from scratch, nor the burden of adapting a tool which was not designed to do so.

1.3 Applications

At present, OPEN/CÆSAR has been used to develop various utilities:

- a small interactive simulator, with back-tracking;
- a verification tool based on exhaustive simulation;
- a verification tool generating, for a given LOTOS specification, its graph reduced modulo $\tau^* a$ bisimulation;
- a verification tool performing deadlock detection using Holzmann's technique for state space representation;
- a symbolic model-checker based on Binary Decision Diagrams;

Chapter 2

Architecture of the OPEN/CÆSAR environment

2.1 Modular decomposition

The "standard" CÆSAR tool generates a C program (known as the *simulator* program) which performs exhaustive simulation to generate the graph.

The basic idea of the OPEN/CÆSAR architecture is to separate the functionalities provided by this simulator program into three parts:

- the *graph module*,
- the *storage module*,
- the *exploration module*.

This separation is modular: the functionality of each part is greatly independent from the other.

In "standard" CÆSAR, only reachability analysis is supported, so the three above modules are intricated. OPEN/CÆSAR brings a clear separation into these three parts.

2.2 The graph module

The "graph module" provides a C representation for the states and the labels of the transition system corresponding to the source LOTOS program. It also provides primitives to handle states and labels.

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

The features provided by the graph module obviously depend on the source LOTOS program under consideration. However, these features can be accessed through an interface which does not depend on any particular LOTOS program.

Practically, let's assume in the sequel that the LOTOS program under consideration is contained in a file named "`spec.lotos`".

Let's assume also that the implementation in C of LOTOS sorts and operations defined in "`spec.lotos`" is contained in a file named "`spec.h`" (this file may have been either written by hand or generated automatically by CÆSAR.ADT).

The graph module corresponding to "`spec.lotos`" is contained in a file named "`spec.c`", which includes "`spec.h`" (using a `#include` directive).

The C file "`spec.c`" can be generated by any version of CÆSAR greater or equal to 4.3. This is done using the "`-open`" version:

```
caesar -open spec.lotos
```

The features provided by "`spec.c`" are described in a file named "`caesar_graph.h`". This interface is a part of the OPEN/CÆSAR distribution package. It is fully independent from "`spec.lotos`". Any C file "`spec.c`" generated by CÆSAR (for a given source program "`spec.lotos`") constitutes an implementation of the features described in "`caesar_graph.h`".

2.3 The storage module

The storage module provides data structures (and their related primitive operations) in order to store the graph, entirely or partially.

There are potentially a number of data structures which can be used, depending on what is to be stored (the states, the labels, the transition relation, etc.) and the particular storage technique chosen.

For instance, it is foreseen to have at least the following data structures (some of them are already implemented):

- a state table for exhaustive simulation with hash access (as in "standard" CÆSAR);
- a state queue for (pseudo) breadth-first search (as in "standard" CÆSAR);
- a state stack for depth-first search;
- a bitmap table (as described in [Holzmann 90]);
- etc.

New data structures will be added progressively.

By selecting an appropriate data structure, the user can decide how many states — and which — are to be stored in memory. This allows to go beyond reachability analysis.

The storage module also provides features of current use, although they are not — strictly speaking — data structures. For instance, it exports hashing functions.

The storage module is logically independent from the source LOTOS program under consideration.

The storage is divided into components, which are called *libraries*. Each library corresponds to a given data structure, or to a particular implementation for this data structure.

The features provided by a given library are described in an interface file named "`caesar_*.h`", where "`*`" stands for the name of the library.

The implementations of the libraries are available under binary form. The object code for all the libraries is gathered into a single archive file named "`libcaesar.a`".

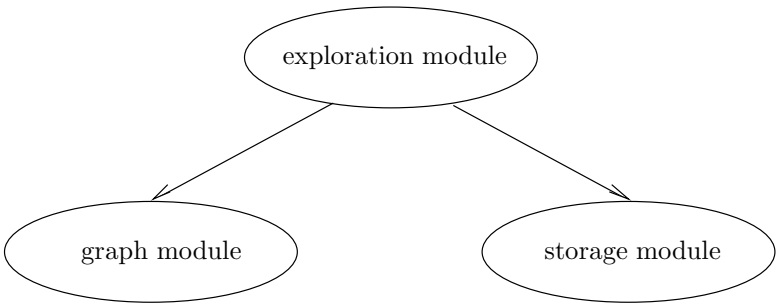

Figure 2.1: Combination of the three modules

2.4 The exploration module

The exploration module can be considered as the main program. It makes use of the graph module and the storage module. This is shown on Figure 2.1.

This module determines how the graph is to be explored. There are many possibilities, depending on various decisions concerning:

- the kind of traversal: depth-first, breadth-first, ...;
- the selection of successor states: how many successor states of the current state — and which – are to be visited? all, some of them, a single one randomly chosen, a single one interactively chosen, ...;
- the storage policy;
- etc.

The exploration module, which contains the "kernel" of the verification algorithm, is usually written by hand. It is expected that, eventually, a set of "exploration" modules will be "standardized" and proposed to the scientific community.

The exploration module should be independent from the source LOTOS program under consideration (so that it can be used with any LOTOS program). However, in some cases, it may depend on the LOTOS source program in order, for instance, to verify a special property.

Let's assume in the sequel that the exploration module is contained in a C file named "`user.c`" provided by the user.

Note: the exploration could be split into several files. This would be useful if the exploration module would be a large program. But this situation is not expected to occur very often.

Note: the exploration module could be written into another language than C, provided that this language allows access to the OPEN/CÆSAR libraries, and provided that link-editing is possible between programs written in this language and C programs. For instance, some exploration modules have already been written in C++.

Some conventions must be followed when writing "`user.c`":

- to use the features of one or several libraries of the storage module, "`user.c`" must include the corresponding interface files, e.g.:

```
#include "caesar_library_1.h"
#include "caesar_library_2.h"
...
#include "caesar_library_n.h"
```

- all identifiers defined in "`user.c`" should start with the prefix "`caesar_`" in order to avoid name conflicts with the identifiers of C types, functions, and macro-definitions defined in "`spec.h`". If this rule is enforced, there can be no conflict between:
  - the identifiers defined in "`user.c`" and those defined in "`spec.h`" , since the latter can not be prefixed by "`caesar_`" (this is forbidden by CÆSAR and CÆSAR.ADT);
  - the identifiers defined in "`user.c`" and those automatically generated by CÆSAR and/or CÆSAR.ADT, since the latter are always prefixed by "`CAESAR_`";
  - the identifiers automatically generated by CÆSAR and those defined in "`spec.h`", since the former are always prefixed by "`CAESAR_`" and the latter can not be prefixed by "`CAESAR_`" (this is forbidden by CÆSAR and CÆSAR.ADT).

  Note: this rule could be relaxed if the "link mode" (see below) is used. In this case, only the non-local objects of "`user.c`" (those searched during link-edit) must be prefixed by "`caesar_`" in order to avoid possible name conflicts. But it is advised that all identifiers start with the prefix "`caesar_`"; this will allow to switch easily between include and link modes.

There are two different ways of combining "`spec.c`" and "`user.c`": the link mode and the include mode.

2.5 The link mode

Using the link mode, "`spec.c`" and "`user.c`" are compiled separately and linked together afterwards.

In this case, "`user.c`" must start with the following directive:

```
#include "caesar_graph.h"
```

where "`caesar_graph.h`" is the interface of the graph module.

Producing the executable program "`user`" can be done using the following UNIX commands:

```
cc -c spec.c
cc -c user.c
cc spec.o user.o -L$CADP/bin.‘arch‘ -lcaesar -o user
```

Note: the "`-L$CADP/bin.‘arch‘ -lcaesar`" options give access to the storage module. They can be omitted if no library of the storage module is used.

Note: the exploration module can be split into several files which are compiled separately and linked together. Each of these files must include the "`caesar_graph.h`" interface.

The overall structure of the link mode is summarized on Figure 2.2. The graphical conventions are the following:

- files are represented by ellipses
- tools (i.e., compilers) are represented by rectangular boxes;

- a plain arrow from some file F to some tool T means that T takes F as input;
- a plain arrow from some tool T to some file F means that T produces F as output;
- a dashed arrow from some file F to another file F' means that F is to be included in F' using a "`#include`" directive;
- a dotted arrow from some file F to another file F' means that F is an interface which is implemented by F'.

2.6 The include mode

Using the include mode, "`spec.c`" and "`user.c`" are concatenated (in order to constitute a new file "`tmp.c`") and compiled together.

Therefore, producing the executable program "`user`" can be done using the UNIX command:

```
cc tmp.c -L$CADP/bin.‘arch‘ -lcaesar -o user
```

Note: the exploration module can be split into several files which are compiled separately and linked together. In this case, "`spec.c`" must be concatenated to one of these files; the remaining files can include "`caesar_graph.h`" if they need access to the graph module features.

The overall structure of the include mode is summarized on Figure 2.3. The graphical conventions are those of Figure 2.2.

2.7 Comparison between link and include modes

The include mode has several advantages:

- the executable program "`user`" generated using the include mode may be slightly more efficient than the one generated using the link mode.

  Indeed, when using the link mode, the actual structure of states and labels is not available in "`user.c`". States and labels are merely represented as generic pointers to byte strings. The length of these strings remains constant during the execution, but it can not be known statically (i.e., when compiling "`user.c`").

  Using the include mode, the actual sizes of states and labels are known statically . This leads to more efficient generated code (for instance, the implementation of a state table is more efficient if the state size is known statically).

  Also, using the include mode, some functions of the graph module are implemented as macro-definitions, for efficiency. This is not possible using the link mode, since these macro-definitions would only be visible in "`spec.c`", but not in "`user.c`".

On the opposite, the link mode has several advantages:

- building "`user`" is faster when "`spec.c`" is already compiled (especially if "`spec.c`" is large);
- the link mode allows "`user.c`" to be written in another language than C;
- the link mode allows to distribute the exploration module under object code form ("`user.o`") not distributing the source code ("`user.c`"). Moreover, binaries written by OPEN/CÆSAR can be protected, if licencing is considered.

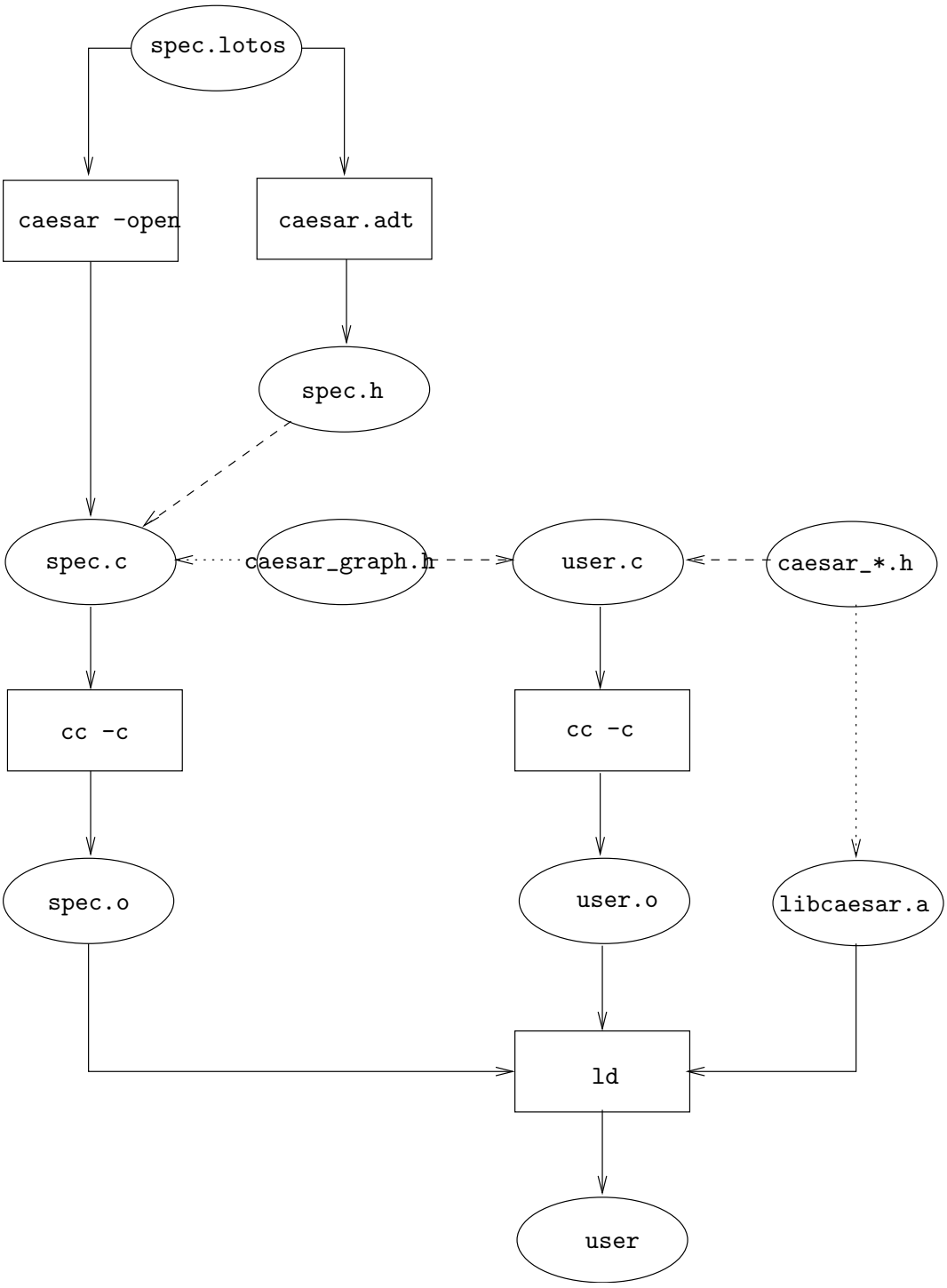

Figure 2.2: Description of the link mode

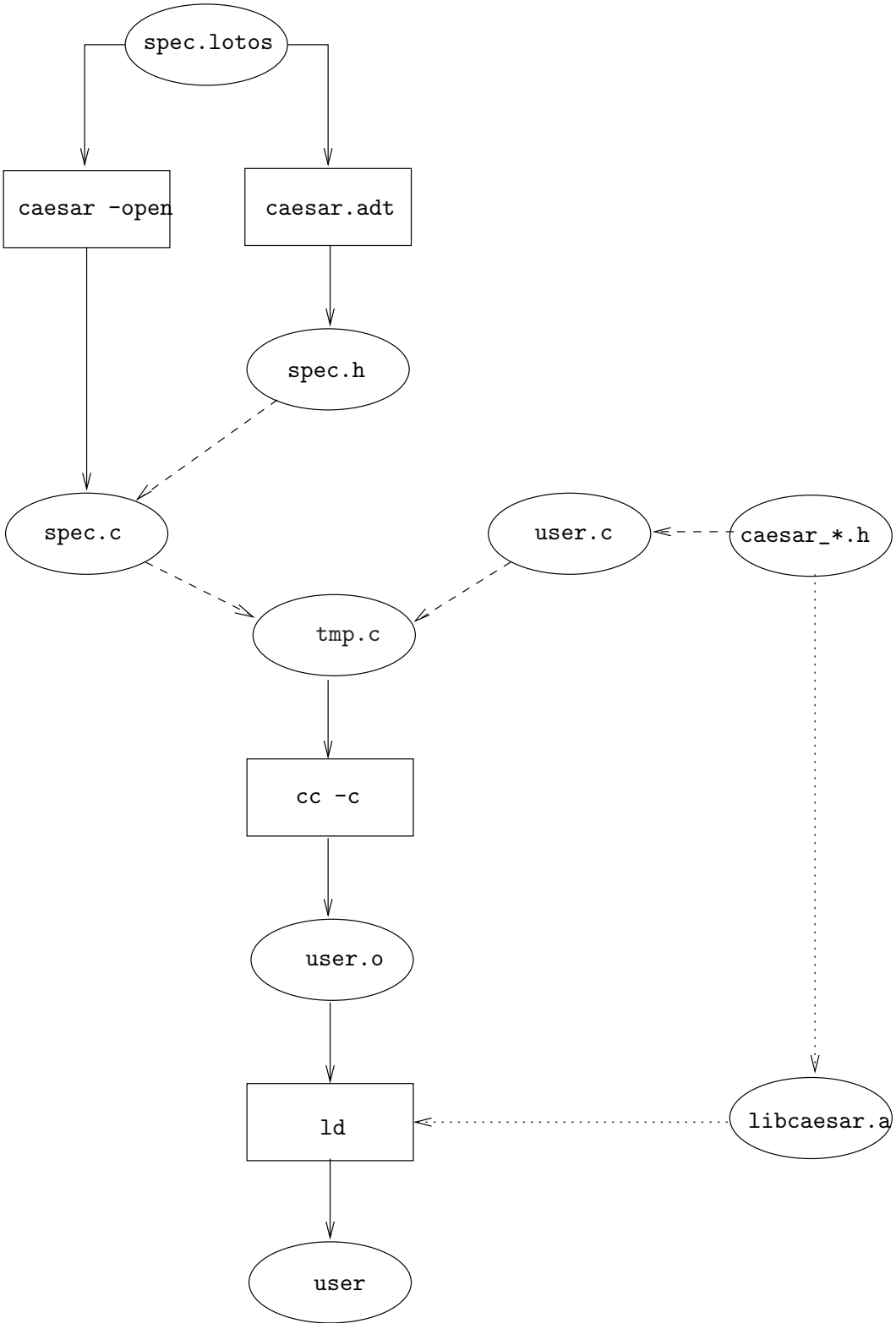

Figure 2.3: Description of the include mode

2.8 The "lotos.open' shell script

The OPEN/CÆSAR environment contains a shell script (named "`lotos.open`) which can be helpful: it determines automatically whether include or link mode can be used, and performs all necessary calls to CÆSAR and the C compiler. Please refer to the "`lotos.open`" manual pages.

Part II

Interfaces

Chapter 3

The "standard" library (version 1.3)

by Hubert Garavel

3.1 Purpose

The "standard" library provides basic types and notations shared by all Open/Cæsar modules.

3.2 Usage

The "standard" library only consists of a predefined header file "`caesar_standard.h`".

3.3 Features

```
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <signal.h>
```

Various standard C libraries are imported, notably those providing functions for dynamic memory allocation, input/output, byte string facilities, and software signals. In particular, the `FILE` type and file handling primitives are imported.

. . . . . .

```
typedef unsigned long CAESAR_TYPE_NATURAL;
```

`CAESAR_TYPE_NATURAL` is the unsigned integer type used in Open/Cæsar.

. . . . . .

```
typedef long CAESAR_TYPE_INTEGER;
```

`CAESAR_TYPE_INTEGER` is the signed integer type used in Open/Cæsar.

```
typedef unsigned char CAESAR_TYPE_BOOLEAN;
```

`CAESAR_TYPE_BOOLEAN` is the boolean type used in Open/Cæsar. It follows the usual conventions of the C language: 0 means "false" and any value different from 0 means "true".

```
#define CAESAR_FALSE ((CAESAR_TYPE_BOOLEAN) 0)
```

`CAESAR_FALSE` is the "false" boolean value.

```
#define CAESAR_TRUE ((CAESAR_TYPE_BOOLEAN) 1)
```

`CAESAR_TRUE` is one possible "true" boolean value.

```
typedef unsigned char CAESAR_TYPE_BYTE;
```

`CAESAR_TYPE_BYTE` is the byte type used in Open/Cæsar.

```
typedef char *CAESAR_TYPE_STRING;
```

`CAESAR_TYPE_STRING` is the string type (pointer to a character string termined by a null character) used in Open/Cæsar.

```
typedef FILE *CAESAR_TYPE_FILE;
```

`CAESAR_TYPE_FILE` is the file type (pointer to a POSIX file descriptor) used in Open/Cæsar.

```
typedef unsigned char *CAESAR_TYPE_POINTER;
```

`CAESAR_TYPE_POINTER` is a pointer to a (generic) byte string.

. . . . . .

```
typedef int CAESAR_TYPE_GENUINE_INT;
```

`CAESAR_TYPE_GENUINE_INT` is a 32-bit integer type; its usage is discouraged, except at specific places where the C standard explicitly requires an "int" type to be used.

. . . . . .

```
typedef CAESAR_TYPE_GENUINE_INT CAESAR_TYPE_ARGC;
```

`CAESAR_TYPE_ARGC` is a 32-bit integer type; it is used to declare the `argc` parameter of the `main()` function of a C program.

. . . . . .

```
typedef char **CAESAR_TYPE_ARGV;
```

`CAESAR_TYPE_ARGV` is a pointer to an array of strings; it is used to declare the `argv` parameter of the `main()` function of a C program.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_SIZE_POINTER ()
   { ... }
```

This function returns the size (in bytes) of a value of type `CAESAR_TYPE_POINTER`.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_ALIGNMENT_POINTER ()
   { ... }
```

This function returns the alignment factor (in bytes) for a value of type `CAESAR_TYPE_POINTER`.

Note: The alignment factor is often dependent from the machine architecture. For any memory area (and not only those of type `CAESAR_TYPE_POINTER`), the alignment factor is always a power of two (1, 2, 4, ...) and is an exact divider of the size of the area. Whatever the machine architecture, any memory area must start at a machine address that is an even multiple of the alignment factor for that area, this constraint stating that areas must be properly aligned on machine word boundaries. In general, one has only to care about alignment constraints when creating structures containing areas with different alignment factors.

. . . . . .

```
typedef void (*CAESAR_TYPE_GENERIC_FUNCTION) ();
```

`CAESAR_TYPE_GENERIC_FUNCTION` is the "pointer to a function" type used in Open/Cæsar. The number of parameters for this function, the types of these parameters (if any) and the result type for this function are not specified.

. . . . . .

```
typedef CAESAR_TYPE_BOOLEAN (*CAESAR_TYPE_COMPARE_FUNCTION)
   (CAESAR_TYPE_POINTER, CAESAR_TYPE_POINTER);
```

`CAESAR_TYPE_COMPARE_FUNCTION` is the "pointer to a comparison function" type used in Open/Cæsar. A comparison function takes two parameters (two pointers to data of the same type) and returns a boolean value, depending whether both pointers refer to identical data or not.

. . . . . .

```
typedef CAESAR_TYPE_NATURAL (*CAESAR_TYPE_HASH_FUNCTION)
   (CAESAR_TYPE_POINTER, CAESAR_TYPE_NATURAL);
```

`CAESAR_TYPE_HASH_FUNCTION` is the "pointer to a hash function" type used in Open/Cæsar. A hash function takes two parameters (a pointer to data and a natural number N) and returns a natural number in the range $0..N-1$.

. . . . . .

```
typedef CAESAR_TYPE_STRING (*CAESAR_TYPE_CONVERT_FUNCTION)
   (CAESAR_TYPE_POINTER);
```

`CAESAR_TYPE_HASH_FUNCTION` is the "pointer to a conversion function" type used in Open/Cæsar. A conversion function takes one parameter (a pointer to data) and returns a character string containing the data under some printable form.

. . . . . .

```
typedef void (*CAESAR_TYPE_PRINT_FUNCTION)
   (CAESAR_TYPE_FILE, CAESAR_TYPE_POINTER);
```

`CAESAR_TYPE_PRINT_FUNCTION` is the "pointer to a printing procedure" type used in Open/Cæsar. A printing procedure takes two parameters (a pointer to a file and a pointer to data) and prints the latter to the former.

. . . . . .

```
typedef unsigned char CAESAR_TYPE_FORMAT;
```

`CAESAR_TYPE_FORMAT` is the type used to control the output of the various printing functions (i.e., functions whose name starts with "`CAESAR_PRINT_`") specified in the Open/Cæsar "graph module" interface or provided by the Open/Cæsar library.

Many of these functions can display a given object under various formats with different degree of information (e.g., a concise format vs a verbose format).

Each format is represented by a small value of type `CAESAR_TYPE_FORMAT`, e.g., 0, 1, 2, etc.

The format to be used by a printing function is not passed to this function as an argument (this would be too heavy); instead, the format is stored either in a global variable or in a field contained in the object to be printed.

.....

```
#define CAESAR_INVALID_FORMAT ((CAESAR_TYPE_FORMAT) 255)
```

`CAESAR_INVALID_FORMAT` is a special value of the type `CAESAR_TYPE_FORMAT`. It can be returned as a result by function `CAESAR_HANDLE_FORMAT()` when this function is invoked with an incorrect argument.

.....

```
#define CAESAR_CURRENT_FORMAT ((CAESAR_TYPE_FORMAT) 254)
```

`CAESAR_CURRENT_FORMAT` is a special value of the type `CAESAR_TYPE_FORMAT`. It can be passed as an argument to function `CAESAR_HANDLE_FORMAT()` and to the various functions whose name starts with "`CAESAR_FORMAT_`" so as to query the current value of the global variable or object field that stores the format used by the corresponding printing function.

.....

```
#define CAESAR_MAXIMAL_FORMAT ((CAESAR_TYPE_FORMAT) 253)
```

`CAESAR_MAXIMAL_FORMAT` is a special value of the type `CAESAR_TYPE_FORMAT`. It can be passed as an argument to function `CAESAR_HANDLE_FORMAT()` and to the various functions whose name starts with "`CAESAR_FORMAT_`" so as to query the largest format value supported by the corresponding printing function.

.....

```
void CAESAR_PRINT_FORMAT (CAESAR_FILE, CAESAR_FORMAT)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This procedure prints to file `CAESAR_FILE` the value of `CAESAR_FORMAT`. If `CAESAR_FORMAT` is equal to one of the three special values `CAESAR_INVALID_FORMAT`, `CAESAR_CURRENT_FORMAT`, or `CAESAR_MAXIMAL_FORMAT`, its value is printed symbolically rather than numerically.

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

```
CAESAR_TYPE_FORMAT CAESAR_HANDLE_FORMAT (CAESAR_VARIABLE, CAESAR_VALUE,
                                         CAESAR_MAXIMAL_VALUE)
   CAESAR_TYPE_FORMAT *CAESAR_VARIABLE;
   CAESAR_TYPE_FORMAT CAESAR_VALUE;
   CAESAR_TYPE_FORMAT CAESAR_MAXIMAL_VALUE;
   { ... }
```

This function is an auxiliary function that should be used to define all the various functions whose name starts with "`CAESAR_FORMAT_`". Examples of the proper use of this function can be found by inspecting the C code generated by Cæsar or Exp.Open.

This function consults or modifies the format value pointed to by `CAESAR_VARIABLE`, which is expected to be the address of the global variable or object field that stores the format used by the corresponding printing function.

If `CAESAR_VALUE` is equal to `CAESAR_CURRENT_FORMAT`, this function returns the current value of the format pointed to by `CAESAR_VARIABLE`.

If `CAESAR_VALUE` is equal to `CAESAR_MAXIMAL_FORMAT`, this function returns `CAESAR_MAXIMAL_VALUE`, which is expected to be the largest format value supported by the corresponding printing function.

If `CAESAR_VALUE` is less or equal to `CAESAR_MAXIMAL_VALUE`, then the value of the format pointed to by `CAESAR_VARIABLE` is modified and set to `CAESAR_VALUE`. In such case, this function returns an undefined value.

In all other cases, this function returns `CAESAR_INVALID_FORMAT`.

```
#define CAESAR_TYPE_ABSTRACT(CAESAR_NAME) struct CAESAR_NAME *
```

`CAESAR_TYPE_ABSTRACT (CAESAR_NAME)` is a pointer to a structure of name `CAESAR_NAME`.

Usually, the definition of `CAESAR_NAME` is not available, so that `CAESAR_TYPE_ABSTRACT (...)` is a "generic" pointer to some data structure, the internal representation of which is not pertinent to the user. This data structure is "abstract", in the sense that one should not rely on a particular implementation.

If `CAESAR_X` is an expression of type `CAESAR_TYPE_ABSTRACT (...)`, dereferencing `CAESAR_X` (e.g., "`*CAESAR_X`" or "`CAESAR_X->...`") is not allowed.

Note: if `CAESAR_X` is an expression of type `CAESAR_TYPE_ABSTRACT (...)`, its value usually obeys alignment constraints (for instance, it can be an address multiple of 4).

```
#define CAESAR_CREATE(CAESAR_ADDRESS,CAESAR_SIZE,CAESAR_TYPE) \
        (CAESAR_ADDRESS) = (CAESAR_TYPE) malloc (CAESAR_SIZE)
```

This macro-definition encapsulates the C function `malloc(3)`. It allocates a memory area of `CAESAR_SIZE` bytes and assigns its address to variable `CAESAR_ADDRESS`, which should be of type

CAESAR_TYPE. If allocation fails, a NULL pointer is assigned to CAESAR_ADDRESS.

.....

```
#define CAESAR_DELETE(CAESAR_ADDRESS) \
        free ((char *) (CAESAR_ADDRESS))
```

This macro-definition encapsulates the C function free(3). It frees the memory area pointed to by CAESAR_ADDRESS.

.....

```
extern CAESAR_TYPE_STRING CAESAR_TOOL;
```

The global variable CAESAR_TOOL is a pointer to a character string which should contain the name of the OPEN/CÆSAR application program. By default, CAESAR_TOOL is always initialized to the empty string "". Although this default value can be left unchanged, it is advisable for each application program to give a meaningful value to CAESAR_TOOL. This is generally done by the first instruction of the main() routine:

```
int main (argc, argv)
int  argc;
char *argv[];
{
/* additional declarations */
CAESAR_TOOL = argv[0];
/* other instructions */
}
```

.....

```
void CAESAR_WARNING (CAESAR_TYPE_STRING CAESAR_FORMAT, ...)
   { ... }
```

This function displays a (non-fatal) warning message to the standard output. The warning message is specified by the actual parameters passed to the function. These parameters follow the same conventions as for printf(3) and their number can be variable. The warning message will be prefixed by the value of the global variable CAESAR_TOOL, unless this value is the empty string. The format string given by CAESAR_FORMAT should not be suffixed with an end-of-line character, as CAESAR_WARNING() will add one automatically.

.....

```
void CAESAR_ERROR (CAESAR_TYPE_STRING CAESAR_FORMAT, ...)
   { ... }
```

This function displays a (fatal) error message to the standard output and stops using exit (1). The error message is specified by the actual parameters passed to the function. These parameters follow

the same conventions as for `printf(3)` and their number can be variable. The error message will be prefixed by the value of the global variable `CAESAR_TOOL`, unless this value is the empty string. The format string given by `CAESAR_FORMAT` should not be suffixed with an end-of-line character, as `CAESAR_ERROR()` will add one automatically.

.....

```
#define CAESAR_PROTEST() ...
```

This macro-definition displays an error message (containing the current file name and current line number) and stops. This macro-definition should be used only to report about unexpected, internal errors. If a more detailed error message should be displayed, then function `CAESAR_ERROR()` should be used instead.

.....

```
#define CAESAR_ASSERT(CAESAR_ASSERTION) \
        { if (!(CAESAR_ASSERTION)) CAESAR_PROTEST(); }
```

This macro-definition evaluates the boolean expression `CAESAR_ASSERTION` and, if the result is false, displays an error message (containing the current file name and current line number) and stops. This macro-definition should be used only to check assertions and report about unexpected, internal errors. If a more detailed error message should be displayed, then function `CAESAR_ERROR()` should be used instead.

.....

```
typedef void (*CAESAR_TYPE_SIGNAL_HANDLER) (int);
```

`CAESAR_TYPE_SIGNAL_HANDLER` is the "pointer to a signal-handler procedure" type used in Open/Cæsar. A signal-handler procedure takes one parameter, which a POSIX signal number. The type `CAESAR_TYPE_SIGNAL_HANDLER` is identical to the type `sighandler_t` that exists in certain Unix distributions. See the "`signal`" manual page for further details.

.....

```
void CAESAR_SET_SIGNALS (CAESAR_HANDLER)
   CAESAR_TYPE_SIGNAL_HANDLER CAESAR_HANDLER;
   { ... }
```

This procedure sets the POSIX signal handler to `CAESAR_HANDLER` for the following signals: `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGILL`, `SIGABRT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, `SIGTERM`, and `SIGPIPE` (whenever these signals are supported by the operating system implementation). In particular, the signal handler `CAESAR_HANDLER` can be equal to the predefined values `SIG_IGN` or `SIG_DFL`. See the "`signal`" manual page for further details.

. . . . . .

```
void CAESAR_RESET_SIGNALS ()
   { ... }
```

This procedure resets the POSIX signal handler for the following signals: `SIGHUP`, `SIGINT`, `SIGQUIT`, `SIGILL`, `SIGABRT`, `SIGFPE`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, `SIGTERM`, and `SIGPIPE` (whenever these signals are supported by the operating system implementation). If the procedure `CAESAR_SET_SIGNALS` has been invoked (at least once) and if no invocation of `CAESAR_RESET_SIGNALS` occurred after the most recent invocation of `CAESAR_SET_SIGNALS`, this signal handler is reset to the value of the `CAESAR_HANDLER` argument passed to the most recent invocation of `CAESAR_SET_SIGNALS`. If the procedure `CAESAR_SET_SIGNALS` has never been invoked or not been invoked after the most recent invocation of `CAESAR_RESET_SIGNALS`, this signal handler is reset to `SIG_DFL`.

. . . . . .

```
CAESAR_TYPE_STRING CAESAR_TEMPORARY_FILE (CAESAR_TEMPORARY_SUFFIX)
   CAESAR_TYPE_STRING CAESAR_TEMPORARY_SUFFIX;
   { ... }
```

This function returns a character string that can safely be used as a file name for a temporary file. The prefix of this file name (i.e., the directory in which the temporary file will be created) is either given by the environment variable `$CADP_TMP` if this variable is defined, or is equal to `"/tmp"` otherwise. The suffix of this file name is given by `CAESAR_TEMPORARY_SUFFIX`, which is usually a file extension starting with a dot. If the allocation of the file name fails (due to a lack of memory) or if the temporary file cannot not be created in the directory specified by `$CADP_TMP` (e.g., because this directory does not exist, is write-protected, or belongs to a file system with unsufficient disk space to create new files), the `NULL` value is returned.

The contents of the character string returned by `CAESAR_TEMPORARY_FILE()` differs at every call to this function.

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

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

. . . . . .

```
void CAESAR_OPEN_COMPRESSED_FILE (CAESAR_FILE, CAESAR_FILENAME, CAESAR_MODE)
   CAESAR_TYPE_FILE *CAESAR_FILE;
   CAESAR_TYPE_STRING CAESAR_FILENAME;
   CAESAR_TYPE_STRING CAESAR_MODE;
   { ... }
```

This function opens (for reading or writing) a file, the pathname of which is `CAESAR_FILENAME`. The value of `CAESAR_MODE`, which must be equal to `"r"` or `"w"`, determines whether this file is opened for reading (`"r"`) or writing (`"w"`).

If the suffix of CAESAR_FILENAME corresponds to a known compression format (".Z", ".gz", ".bz2", etc.), the file will be treated as a compressed file. Otherwise, the file will not be compressed and treated as an ordinary file. The list of supported suffixes and compression formats is given by the cadp_zip shell-script provided within the CADP distribution.

If the file can be opened, the value of *CAESAR_FILE is set to a POSIX stream descriptor from (respectively, to) which data can be read (respectively, written) using the standard input (respectively, output) routines available in POSIX, e.g., fgets(3), fprintf(3), fputs(3), fscanf(3), etc. Data compression, if any, is performed transparently, meaning that the data sent or read to a compressed file are exactly the same as if the file was not compressed.

If the file cannot be opened, the value of *CAESAR_FILE is set to NULL.

Note: Since type CAESAR_TYPE_FILE is defined as FILE *, variable CAESAR_FILE is actually of type FILE **, where FILE is the type of a POSIX file descriptor.

Note: Function CAESAR_OPEN_COMPRESSED_FILE() invokes fopen(3) to open an uncompressed file and popen(3) to open a compressed file.

. . . . . .

```
void CAESAR_CLOSE_COMPRESSED_FILE (CAESAR_FILE)
   CAESAR_TYPE_FILE *CAESAR_FILE;
   { ... }
```

This function closes the file pointed to by *CAESAR_FILE, which must have been opened previously by a call to the CAESAR_OPEN_COMPRESSED_FILE() function. After the file is closed, the value of *CAESAR_FILE is set to NULL.

Note: Since type CAESAR_TYPE_FILE is defined as FILE *, variable CAESAR_FILE is actually of type FILE **, where FILE is the type of a POSIX file descriptor.

Note: Function CAESAR_CLOSE_COMPRESSED_FILE() invokes either fclose(3) or pclose(3) to close the file.

. . . . . .

```
CAESAR_TYPE_STRING CAESAR_FUNCTION_NAME (CAESAR_FUNCTION)
   void (*CAESAR_FUNCTION) ();
   { ... }
```

This function returns a printable character string identifying the function CAESAR_FUNCTION. If CAESAR_FUNCTION belongs to a predefined set of functions of the Open/Cæsar library, the character string contains the name of this function (e.g., "CAESAR_0_HASH", "CAESAR_PRINT_STATE", "CAESAR_COMPARE_LABEL", etc.); otherwise, the character string contains the hexadecimal value of the function pointer CAESAR_FUNCTION.

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

Note: In fact, CAESAR_FUNCTION_NAME() is implemented as a macro-definition that invokes an auxiliary function. This avoids the need for inserting a type cast before any parameter given to CAESAR_FUNCTION_NAME().

Chapter 4

The "version" library (version 2.0)

by Hubert Garavel

4.1 Purpose

The "version" library allows to access and control the version numbers (also called release numbers, revision numbers) of the tools and libraries making up the OPEN/CÆSAR environment.

It is be used to check whether a "`spec.c`" file (generated from a source program) is up to date with respect to the "`caesar_graph,h`" file and other files and libraries of the OPEN/CÆSAR environment. Version clashes between "`spec.c`" and "`caesar_graph,h`" can cause subtle errors very difficult to detect; the "version" library aims at preventing such clashes.

4.2 Usage

The "version" library consists of:

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

Note: The "version" library is a software layer built above the primitives offered by the "standard" library.

4.3 Features

```
typedef double CAESAR_TYPE_VERSION;
```

`CAESAR_TYPE_VERSION` represents a version number, which is a positive floating-point number with a single digit after the decimal point.

. . . . . .

```
CAESAR_TYPE_VERSION CAESAR_LIBRARY_VERSION ()
   { ... }
```

This function returns the version number of the OPEN/CÆSAR environment. This version number covers the "`caesar_graph.h`" file, as well as other "`.h`", "`.c`" and "`.a`" files contained in the OPEN/CÆSAR distribution. All these files are supposed to be mutually up to date and compatible.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_COMPARE_VERSION (CAESAR_V1, CAESAR_V2)
   CAESAR_TYPE_VERSION CAESAR_V1;
   CAESAR_TYPE_VERSION CAESAR_V2;
   { ... }
```

This function returns `CAESAR_TRUE` if both version numbers `CAESAR_V1` and `CAESAR_V2` are identical, or `CAESAR_FALSE` if they are not.

Note: the standard C operator "`==`" should not be used for this purpose, because of problems inherent to the floating-point representation. For instance, version number `1.1` will not be exactly represented as `1.1`, but as `1.10000002384185791`, so it is possible that `1.1` is not strictly equal to `1.1`! Function `CAESAR_COMPARE_VERSION` solves this problem by comparing both version numbers with a limited precision ($\pm 10^{-3}$).

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_MATCH_VERSION (CAESAR_V1, CAESAR_V2)
   CAESAR_TYPE_VERSION CAESAR_V1;
   CAESAR_TYPE_VERSION CAESAR_V2;
   { ... }
```

This function returns `CAESAR_TRUE` if `CAESAR_LIBRARY_VERSION()` is contained in the numeric interval [`CAESAR_V1`, `CAESAR_V2`], or `CAESAR_FALSE` otherwise. `CAESAR_V1` and `CAESAR_V2` can be equal: in this case, the result is `CAESAR_TRUE` iff `CAESAR_LIBRARY_VERSION()` is equal to `CAESAR_V1`.

The parameters `CAESAR_V1` and `CAESAR_V2` delimit the bounds of an interval of acceptable version numbers for the value of `CAESAR_LIBRARY_VERSION()`, meaning that "`spec.c`" can be safely compiled and linked with any revision of the OPEN/CÆSAR library whose version number is between `CAESAR_V1` and `CAESAR_V2` (bounds included).

.....

```
void CAESAR_CHECK_VERSION (CAESAR_V1, CAESAR_V2)
   CAESAR_TYPE_VERSION CAESAR_V1;
   CAESAR_TYPE_VERSION CAESAR_V2;
   { ... }
```

This procedure evaluates the following boolean expression:

```
CAESAR_MATCH_VERSION (CAESAR_V1, CAESAR_V2)
```

and aborts the execution if the result is equal to 0.

Note: This function should be called in the "`spec.c`" program, for instance at the beginning of procedure `CAESAR_INIT_GRAPH()`.

.....

```
void CAESAR_PRINT_VERSION (CAESAR_FILE, CAESAR_V)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_VERSION CAESAR_V;
   { ... }
```

This procedure prints to file `CAESAR_FILE` a character string representing the version number `CAESAR_V`.

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

.....

Chapter 5

The "graph" library (version 2.4)

by Hubert Garavel

5.1 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/CÆSAR 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/CÆSAR application programs that rely on the standard input and standard output (e.g., the OPEN/CÆSAR Interactive Simulator) might not function properly if the graph module performs conflicting accesses to the standard input or the standard output.

5.2 Usage

The graph module consists of:

- A predefined header file "`caesar_graph.h`", which can be found in the OPEN/CÆSAR 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 CÆSAR 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/CÆSAR 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/CÆSAR 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.

5.3 General features

```
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 CÆSAR 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_TYPE_VERSION CAESAR_GRAPH_VERSION ()
   { ... }
```

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

.....

```
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".

. . . . . .

5.4 State features

```
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.

. . . . . .

```
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_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_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:

$$0 < \texttt{CAESAR\_HASH\_SIZE\_STATE ()} \leq \texttt{CAESAR\_SIZE\_STATE ()}$$

but it is possible that:

$$\texttt{CAESAR\_HASH\_SIZE\_STATE ()} < \texttt{CAESAR\_SIZE\_STATE ()}$$

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_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.

.....

```
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.

. . . . . .

```
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.

. . . . . .

```
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_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_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.

```
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`").

  Cæsar-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).

  Cæsar-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 Cæsar). In all other cases, the effect of this function is undefined.

```
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/Cæsar. 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()`.

```
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)`.

```
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)`.

```
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)`.

5.5 Label features

```
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.

.....

```
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_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_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:

$$0 < \texttt{CAESAR\_HASH\_SIZE\_LABEL ()} \leq \texttt{CAESAR\_SIZE\_LABEL ()}$$

but it is possible that:

$$\texttt{CAESAR\_HASH\_SIZE\_LABEL ()} < \texttt{CAESAR\_SIZE\_LABEL ()}$$

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_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.

. . . . . .

```
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.

. . . . . .

```
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_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., "τ").

. . . . . .

```
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., "τ"), 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 "τ", 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_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.

.....

```
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_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_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.

. . . . . .

```
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_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.

. . . . . .

```
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).

Cæsar-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 CÆSAR). In all other cases, the effect of this function is undefined.

. . . . . .

```
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/CÆSAR. 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_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.

CÆSAR-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 CÆSAR with "`-network`" option.

. . . . . .

5.6 Edge features

```
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.

.....

```
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.

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

. . . . . .

5.7 Obsolete features

```
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/CÆSAR application programs.

. . . . . .

```
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/CÆSAR application programs.

. . . . . .

Chapter 6

The "edge" library (version 1.6)

by Hubert Garavel

6.1 Purpose

The "edge" library provides primitives for computing the edges going out of a given state.

Although the OPEN/CÆSAR graph module already provides an iterator macro for this purpose (the `CAESAR_ITERATE_STATE()` function), higher-level primitives may be useful and easier to use. This is especially the case when a depth-first traversal of the state graph is necessary (e.g., on-the-fly verification, interactive simulation, ...).

From our experience, the `CAESAR_ITERATE_STATE()` function is often used as follows: for a given state S_1, one wants to compute all outgoing edges of the form "(S_1, L, S_2)"; the labels L and states S_2 are stored in a data structure (usually a linked list).

6.2 Usage

The "edge" library consists of:

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

Note: The "edge" library is a software layer built above the primitives offered by the "standard" library and by the OPEN/CÆSAR graph module.

6.3 Description

An "edge" is basically a tuple with 5 fields:

- (1) a field containing a "previous" state;

- (2) a field containing a label;
- (3) a field containing a "next" state;
- (4) a "mark" field, that is a byte string whose size and contents are freely determined by the user. It can be used to mark states while depth-first exploring the state graph. The size of the mark field is the same for all edges; it must be greater or equal than zero. Pointers to mark fields will be considered as values of type `CAESAR_TYPE_POINTER`; "mark" fields are always aligned on appropriate boundaries so that the user can put any information in these fields without alignment problem;
- (5) a pointer to a "successor" edge, which is used to build linked lists of edges.

Fields (1), (2), (3), and (4) are optional, depending on the initialization parameters (see function `CAESAR_INIT_EDGE()` below).

Edges are represented as byte strings of fixed size (see function `CAESAR_SIZE_EDGE()` below) with definite alignment constraints (see function `CAESAR_ALIGNMENT_EDGE()` below). All edges have the same size.

6.4 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_EDGE;
```

This type denotes a pointer to the concrete representation of an edge. The edge representation is supposed to be "opaque".

.....

```
void CAESAR_INIT_EDGE (CAESAR_PREVIOUS_STATE, CAESAR_LABEL, CAESAR_NEXT_STATE,
                       CAESAR_SIZE_MARK, CAESAR_ALIGNMENT_MARK)
CAESAR_TYPE_BOOLEAN CAESAR_PREVIOUS_STATE;
CAESAR_TYPE_BOOLEAN CAESAR_LABEL;
CAESAR_TYPE_BOOLEAN CAESAR_NEXT_STATE;
CAESAR_TYPE_NATURAL CAESAR_SIZE_MARK;
CAESAR_TYPE_NATURAL CAESAR_ALIGNMENT_MARK;
{ ... }
```

This initialization procedure must be called before using any other primitive of the "edge" library. It should be called only once.

Each edge will contain a "previous" state iff `CAESAR_PREVIOUS_STATE` is equal to true.

Each edge will contain a label iff `CAESAR_LABEL` is equal to true.

Each edge will contain a "next" state iff `CAESAR_NEXT_STATE` is equal to true.

Each edge will contain a mark field iff `CAESAR_SIZE_MARK` is different from zero. If so, the value of `CAESAR_SIZE_MARK` determines the (constant) size (in bytes) of the mark field, and the value of `CAESAR_ALIGNMENT_MARK` determines the alignment factor (in bytes) of the mark field. The alignment factor must be a power of two. Any mark field will be aligned on a boundary that is an even multiple of the alignment factor. `CAESAR_ALIGNMENT_MARK` is equal to zero iff `CAESAR_SIZE_MARK` is equal to zero; otherwise, the effect of `CAESAR_INIT_EDGE()` is undefined.

If CAESAR_PREVIOUS_STATE, CAESAR_LABEL, and CAESAR_NEXT_STATE are equal to false, and if CAESAR_SIZE_MARK is equal to zero, the effect of CAESAR_INIT_EDGE() is undefined.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_SIZE_EDGE ()
{ ... }
```

This function returns the edge size (in bytes).

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_ALIGNMENT_EDGE ()
   { ... }
```

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

. . . . . .

```
CAESAR_TYPE_STATE CAESAR_PREVIOUS_STATE_EDGE (CAESAR_E)
CAESAR_TYPE_EDGE CAESAR_E;
{ ... }
```

This function returns a pointer to the "previous" state field of the edge pointed to by CAESAR_E. If there is no such field (due to the initialization parameters supplied to CAESAR_INIT_EDGE()) the result is undefined.

. . . . . .

```
CAESAR_TYPE_LABEL CAESAR_LABEL_EDGE (CAESAR_E)
CAESAR_TYPE_EDGE CAESAR_E;
{ ... }
```

This function returns a pointer to the label field of the edge pointed to by CAESAR_E. If there is no such field (due to the initialization parameters supplied to CAESAR_INIT_EDGE()) the result is undefined.

. . . . . .

```
CAESAR_TYPE_STATE CAESAR_NEXT_STATE_EDGE (CAESAR_E)
CAESAR_TYPE_EDGE CAESAR_E;
{ ... }
```

This function returns a pointer to the "next" state field of the edge pointed to by CAESAR_E. If there is no such field (due to the initialization parameters supplied to CAESAR_INIT_EDGE()) the result is

undefined.

. . . . . .

```
CAESAR_TYPE_POINTER CAESAR_MARK_EDGE (CAESAR_E)
CAESAR_TYPE_EDGE CAESAR_E;
{ ... }
```

This function returns a pointer to the mark field of the edge pointed to by `CAESAR_E`. If there is no such field (due to the initialization parameters supplied to `CAESAR_INIT_EDGE()`) the result is undefined.

. . . . . .

```
CAESAR_TYPE_EDGE *CAESAR_SUCCESSOR_EDGE (CAESAR_E)
CAESAR_TYPE_EDGE CAESAR_E;
{ ... }
```

This function returns a pointer to the "successor" edge (pointer) field of the edge pointed to by `CAESAR_E`.

. . . . . .

```
void CAESAR_CREATE_EDGE (CAESAR_E)
   CAESAR_TYPE_EDGE *CAESAR_E;
   { ... }
```

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

When the allocation succeeds, the mark field (if any) of `CAESAR_E` is initialized to a bit string of 0's and the "successor" edge field is initialized to the `NULL` pointer. The state field and the label field are left undefined.

Note: because `CAESAR_TYPE_EDGE` is a pointer type, any variable `CAESAR_E` of type `CAESAR_TYPE_EDGE` must be allocated before used, for instance using:

```
CAESAR_CREATE_EDGE (&CAESAR_E);
```

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

. . . . . .

```
void CAESAR_DELETE_EDGE (CAESAR_E)
   CAESAR_TYPE_EDGE *CAESAR_E;
   { ... }
```

This procedure frees the byte string of length `CAESAR_SIZE_EDGE()` pointed to by `*CAESAR_E` using `CAESAR_DELETE()`. Afterwards, the `NULL` value is assigned to `*CAESAR_E`.

```
void CAESAR_COPY_EDGE (CAESAR_E1, CAESAR_E2)
   CAESAR_TYPE_EDGE CAESAR_E1;
   CAESAR_TYPE_EDGE CAESAR_E2;
   { ... }
```

This procedure copies the edge pointed to by `CAESAR_E2` onto the edge pointed to by `CAESAR_E1`.

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_EDGE (CAESAR_FORMAT)
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which edges are printed by procedures `CAESAR_PRINT_EDGE()` and `CAESAR_PRINT_EDGE_LIST()` (see below). Currently, the following formats are available:

- With format 0, the edge is printed as a portion of text. This is mainly intended for debugging purpose.
- (no other format available yet)

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

When called with `CAESAR_FORMAT` between 0 and 0, this fonction sets the current edge format to `CAESAR_FORMAT` and returns an undefined result.

When called with another value of `CAESAR_FORMAT`, this function does not modify the current edge 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 edge format. If `CAESAR_FORMAT` is equal to the constant `CAESAR_MAXIMAL_FORMAT`, the result is the maximal format value (i.e., 0). In all other cases, the effect of this function is undefined.

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_EDGE ()
   { ... }
```

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

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

```
void CAESAR_PRINT_EDGE (CAESAR_FILE, CAESAR_E)
   CAESAR_TYPE_FILE CAESAR_FILE;
```

```
CAESAR_TYPE_EDGE CAESAR_E;
{ ... }
```

This procedure prints to file `CAESAR_FILE` information about the contents of the edge pointed to by `CAESAR_E`. The nature of the information is determined by the current edge format (see procedure `CAESAR_FORMAT_EDGE()` above).

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

.....

```
void CAESAR_CREATE_EDGE_LIST (CAESAR_S1, CAESAR_E1_En, CAESAR_ORDER)
CAESAR_TYPE_STATE CAESAR_S1;
CAESAR_TYPE_EDGE *CAESAR_E1_En;
CAESAR_TYPE_NATURAL CAESAR_ORDER;
```

This procedure computes all couples (`CAESAR_L`, `CAESAR_S2`) such that "(`CAESAR_S1`, `CAESAR_L`, `CAESAR_S2`)" is an edge of the labelled transition system (this is done using the `CAESAR_ITERATE_STATE()` procedure of the graph module).

This procedure also builds a linked list whose items are values of type `CAESAR_TYPE_EDGE`, linked together using the "successor" edge field. The "successor" edge field of the last item is set to `NULL`. The list can be empty if `CAESAR_S1` is a sink state. The address of the first item of the list (or `NULL` if the list is empty) is assigned to `*CAESAR_E1_En`. Obviously, the previous value of `*CAESAR_E1_En` is lost.

The fields of each item are assigned as follows:

- the "previous" state field (if any) will contain the value of `CAESAR_S1`.
- the label field (if any) will contain various values for `CAESAR_L`.
- the "next" state field (if any) of each item will contain values for `CAESAR_S2`.
- the mark field (if any) is initialized to a bit string of 0's.

  The value of the formal parameter `CAESAR_ORDER` determines the order of the items of the linked list. Several cases are currently implemented:

- if `CAESAR_ORDER` is equal to 0, the list order is undefined.
- if `CAESAR_ORDER` is equal to 1, the edge list is sorted in the same order as transitions are enumerated by the `CAESAR_ITERATE_STATE()` procedure.
- if `CAESAR_ORDER` is equal to 2, the edge list is sorted in the reverse order of the order in which transitions are enumerated by the `CAESAR_ITERATE_STATE()` procedure.
- if `CAESAR_ORDER` is equal to 3 or 5, the list is sorted in such a way that the character string values returned by `CAESAR_STRING_LABEL()` are increasing, according to the lexicographical order used in the function `strcmp(3)`.
- if `CAESAR_ORDER` is equal to 4 or 6, the list is sorted in such a way that the character string values returned by `CAESAR_STRING_LABEL()` are decreasing, according to the lexicographical order used in the function `strcmp(3)`.

Additionally, this procedure sets two global variables `caesar_creation` and `caesar_truncation` of type `CAESAR_TYPE_NATURAL`. After any call to `CAESAR_CREATE_EDGE_LIST()`, these variables can be inspected using the two functions `CAESAR_CREATION_EDGE_LIST()` and `CAESAR_TRUNCATION_EDGE_LIST()` defined below. The values of these variables are set as follows:

- if the computation normally succeeds, then `caesar_creation` is set to the number of items in the linked list and `caesar_truncation` is set to zero.
- if allocation fails when building the list (due to a lack of memory), a truncated list is built (the "successor" edge field of the last item is still set to `NULL`). Then `caesar_creation` is set to the number of items in the truncated list and `caesar_truncation` is set to the number of items that have not been inserted in the list (this number is greater than zero).

.....

```
CAESAR_TYPE_NATURAL CAESAR_MAX_ORDER_EDGE_LIST ()
   { ... }
```

This function returns the highest order available for edge list creation, i.e., the highest acceptable value for the parameter `CAESAR_ORDER` of function `CAESAR_CREATE_EDGE_LIST()`.

.....

```
void CAESAR_DELETE_EDGE_LIST (CAESAR_E1_En)
CAESAR_TYPE_EDGE *CAESAR_E1_En;
{ ... }
```

This procedure frees each item of the linked list pointed to by `*CAESAR_E1_En`. Afterwards, the `NULL` value is assigned to `*CAESAR_E1_En`.

.....

```
void CAESAR_COPY_EDGE_LIST (CAESAR_E1_Em, CAESAR_E1_En)
CAESAR_TYPE_EDGE *CAESAR_E1_Em;
CAESAR_TYPE_EDGE CAESAR_E1_En;
{ ... }
```

This procedure builds a duplicate list, which is a copy of the linked list pointed to by `CAESAR_E1_En`. A pointer to this duplicate list (or NULL if the list is empty) is assigned to `*CAESAR_E1_Em`. For each item of the linked list pointed to by `CAESAR_E1_En`, a duplicated item is allocated. Said differently, both lists do not have shared items in common.

Additionally, this procedure sets two global variables `caesar_creation` and `caesar_truncation` of type `CAESAR_TYPE_NATURAL`. After any call to `CAESAR_COPY_EDGE_LIST()`, these variables can be inspected using the two functions `CAESAR_CREATION_EDGE_LIST()` and `CAESAR_TRUNCATION_EDGE_LIST()` defined below. The values of these variables are set as in the `CAESAR_CREATE_EDGE_LIST()` function.

The previous value of `*CAESAR_E1_Em` is lost: if it points to a non-empty list, this list should be freed using `CAESAR_DELETE_EDGE_LIST()` before `CAESAR_COPY_EDGE_LIST()` is called.

```
CAESAR_TYPE_NATURAL CAESAR_CREATION_EDGE_LIST ()
   { ... }
```

This function returns the value of the global variable `caesar_creation` computed during the last call to `CAESAR_CREATE_EDGE_LIST()` or `CAESAR_COPY_EDGE_LIST()`. This variable can only be accessed using this function.

```
CAESAR_TYPE_NATURAL CAESAR_TRUNCATION_EDGE_LIST ()
   { ... }
```

This function returns the value of the global variable `caesar_truncation` computed during the last call to `CAESAR_CREATE_EDGE_LIST()` or `CAESAR_COPY_EDGE_LIST`. This variable can only be accessed using this function.

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_EDGE_LIST (CAESAR_FORMAT)
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which edge lists are printed by the procedure `CAESAR_PRINT_EDGE_LIST()` (see below). Currently, the following formats are available:

- With format 0, the edge list is printed as a portion of text. This is mainly intended for debugging purpose.
- (no other format available yet)

A call to this function sets the current edge list format to `CAESAR_FORMAT`.

When called with `CAESAR_FORMAT` between 0 and 0, this fonction sets the current edge list format to `CAESAR_FORMAT` and returns an undefined result.

When called with another value of `CAESAR_FORMAT`, this function does not modify the current edge list 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 edge list format. If `CAESAR_FORMAT` is equal to the constant `CAESAR_MAXIMAL_FORMAT`, the result is the maximal format value (i.e., 0). In all other cases, the effect of this function is undefined.

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_EDGE_LIST ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as it might be removed from future versions of the Open/Cæsar. Use function `CAESAR_FORMAT_EDGE_LIST()` instead, called

with argument CAESAR_MAXIMAL_FORMAT.

This function returns the maximal format value available for printing edge lists.

.....

```
void CAESAR_PRINT_EDGE_LIST (CAESAR_FILE, CAESAR_E1_En)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_EDGE CAESAR_E1_En;
   { ... }
```

This procedure prints to file CAESAR_FILE information about the contents of the linked list of edges pointed to by CAESAR_E1_En. The nature of the information is determined by the current edge format and the current edge list format (see procedures CAESAR_FORMAT_EDGE() and CAESAR_FORMAT_EDGE_LIST() above).

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

.....

```
#define CAESAR_ITERATE_PLNM_EDGE_LIST(CAESAR_E1_En,CAESAR_E,\
                    CAESAR_S1,CAESAR_L,CAESAR_S2,CAESAR_M) ...
```

with parameters typed as follows:

```
CAESAR_TYPE_EDGE CAESAR_E1_En;
CAESAR_TYPE_EDGE CAESAR_E;
CAESAR_TYPE_STATE CAESAR_S1;
CAESAR_TYPE_LABEL CAESAR_L;
CAESAR_TYPE_STATE CAESAR_S2;
CAESAR_TYPE_POINTER CAESAR_M;
```

This macro-definition is an iterator which can be used in the same way as a "`while (...)`" or "`for (...; ...; ...)`" instruction. It is therefore possible to write an instruction such as:

```
        CAESAR_ITERATE_PLNM_EDGE_LIST(caesar_e1_en,caesar_e,
                    caesar_s1,caesar_l,caesar_s2,caesar_m) {
        ...
        body of the loop, containing occurrences of variables
        caesar_e, caesar_s1, caesar_l, caesar_s2, and caesar_m
        ...
        }
```

CAESAR_E1_En is an expression (r-value) containing a pointer to the first item of a linked list of edges.

CAESAR_E is a variable (l-value) which will be used as the induction variable in the body of the loop. At the n^{th} iteration step, it points to the n^{th} item of the linked list.

CAESAR_S1 is a variable (l-value) which can also be used as an induction variable. At the n^{th} iteration step, it points to the "previous" state field of the n^{th} item of the linked list. If this field does not exist, the result is undefined. At each step, one has:

$$\texttt{CAESAR\_S1} == \texttt{CAESAR\_PREVIOUS\_STATE\_EDGE (CAESAR\_E)}$$

`CAESAR_L` is a variable (l-value) which can also be used as an induction variable. At the n^{th} iteration step, it points to the label field of the n^{th} item of the linked list. If this field does not exist, the result is undefined. At each step, one has:

`CAESAR_L` == `CAESAR_LABEL_EDGE (CAESAR_E)`

`CAESAR_S2` is a variable (l-value) which can also be used as an induction variable. At the n^{th} iteration step, it points to the "next" state field of the n^{th} item of the linked list. If this field does not exist, the result is undefined. At each step, one has:

`CAESAR_S2` == `CAESAR_NEXT_STATE_EDGE (CAESAR_E)`

`CAESAR_M` is a variable (l-value) which can also be used as an induction variable. At the n^{th} iteration step, it points to the mark field of the n^{th} item of the linked list. If this field does not exist, the result is undefined. At each step, one has:

`CAESAR_M` == `CAESAR_MARK_EDGE (CAESAR_E)`

The body of the loop can be any statement of the C language. In particular, it may contain "`break`" and "`continue`" statements with their usual semantics.

This is the most general iterator on linked lists of edges. There are also 15 other iterators derived from the general one. These iterators are simpler than the general one, since they deal with the cases where one or several of the following parameters:

`CAESAR_S1`, `CAESAR_L`, `CAESAR_S2`, `CAESAR_M`

are omitted. These operators are used according to the needs (for example, the four aforementioned parameters can be omitted if one only wants to compute the length of an edge list) and also depending on the initialization values given to `CAESAR_INIT_EDGE()` (since one or several fields may not actually exist).

The 15 derived iterators are listed below.

```
#define CAESAR_ITERATE_EDGE_LIST(CAESAR_E1_En,CAESAR_E) ...

#define CAESAR_ITERATE_P_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_S1) ...

#define CAESAR_ITERATE_L_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_L) ...

#define CAESAR_ITERATE_N_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_S2) ...

#define CAESAR_ITERATE_M_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_M) ...

#define CAESAR_ITERATE_PL_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_S1,CAESAR_L) ...

#define CAESAR_ITERATE_PN_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_S1,CAESAR_S2) ...

#define CAESAR_ITERATE_PM_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_S1,CAESAR_M) ...

#define CAESAR_ITERATE_LN_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_L,CAESAR_S2) ...
```

```
#define CAESAR_ITERATE_LM_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_L,CAESAR_M) ...

#define CAESAR_ITERATE_NM_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_S2,CAESAR_M) ...

#define CAESAR_ITERATE_PLN_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_S1,CAESAR_L,CAESAR_S2) ...

#define CAESAR_ITERATE_PLM_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_S1,CAESAR_L,CAESAR_M) ...

#define CAESAR_ITERATE_PNM_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_S1,CAESAR_S2,CAESAR_M) ...

#define CAESAR_ITERATE_LNM_EDGE_LIST(CAESAR_E1_En,CAESAR_E,CAESAR_L,CAESAR_S2,CAESAR_M) ...
```

.....

```
CAESAR_TYPE_NATURAL CAESAR_LENGTH_EDGE_LIST (CAESAR_E1_En)
   CAESAR_TYPE_EDGE CAESAR_E1_En;
   { ... }
```

This function returns the number of items in the linked list pointed to by `CAESAR_E1_En`.

.....

```
CAESAR_TYPE_EDGE CAESAR_ITEM_EDGE_LIST (CAESAR_E1_En, CAESAR_N)
   CAESAR_TYPE_EDGE CAESAR_E1_En;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns the `CAESAR_N`$^{\text{th}}$ item in the linked list pointed to by `CAESAR_E1_En` (the first item is numbered 1). If `CAESAR_N` is equal to 0, or is greater than the actual length of the linked list, the result is undefined.

.....

```
void CAESAR_REVERSE_EDGE_LIST (CAESAR_E1_En)
   CAESAR_TYPE_EDGE *CAESAR_E1_En;
   { ... }
```

This procedure reverses the linked list of edges pointed to by `*CAESAR_E1_En`.

.....

Chapter 7

The "stack_1" library (version 1.6)

by Hubert Garavel

7.1 Purpose

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

7.2 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/CÆSAR 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.

7.3 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.

7.4 Features

```
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".

.....

```
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.

.....

```
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.

```
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.

```
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.

```
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.

```
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`.

. . . . . .

```
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`.

. . . . . .

```
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()`.

. . . . . .

```
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_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_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_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_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_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_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_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
```

. . . . . .

```
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.

.....

```
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.

.....

```
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`.

. . . . . .

```
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.

. . . . . .

```
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.

. . . . . .

```
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_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 fonction 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_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/Cæsar. Use function `CAESAR_FORMAT_STACK_1()` instead, called with argument `CAESAR_MAXIMAL_FORMAT`.

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

.....

```
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)`.

.....

7.5 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);
}
```

Chapter 8

The "hash" library (version 1.3)

by Hubert Garavel

8.1 Purpose

The "hash" library provides primitives for computing various hashing functions on byte strings, states and labels.

8.2 Usage

The "hash" library consists of:

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

Note: The "hash" library is a software layer built above the primitives offered by the "standard" library and by the OPEN/CÆSAR graph module.

8.3 Features

```
CAESAR_TYPE_BOOLEAN CAESAR_PRIME_HASH (CAESAR_I)
  CAESAR_TYPE_NATURAL CAESAR_I;
  { ... }
```

This function returns a value different from 0 if `CAESAR_I` is a prime number, or 0 otherwise.

.....

```
CAESAR_TYPE_NATURAL CAESAR_0_HASH (CAESAR_P, CAESAR_SIZE, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
```

```
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the byte string of length `CAESAR_SIZE` pointed to by `CAESAR_P`, and returns this value, which is in the range 0..(`CAESAR_MODULUS` − 1). If either `CAESAR_SIZE` or `CAESAR_MODULUS` is equal to 0, the result is undefined.

Note: The string is exactly `CAESAR_SIZE` bytes long, starting from `CAESAR_P`[0] to `CAESAR_P`[`CAESAR_SIZE` − 1]. It is not required that the byte string is terminated by a null byte (which is not the case of character strings in C). The same remark applies to the other functions defined below.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_1_HASH (CAESAR_P, CAESAR_SIZE, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the byte string of length `CAESAR_SIZE` pointed to by `CAESAR_P`, and returns this value, which is in the range 0..(`CAESAR_MODULUS` − 1). If either `CAESAR_SIZE` or `CAESAR_MODULUS` is equal to 0, the result is undefined.

Note: This function is derived from the hash function "`s_hash()`" formerly used in the SPIN validation system (see Gerard Holzmann's book "Design and Validation of Computer Protocols", 1991, page 307). The result of "`s_hash()`" is also the same as the one returned in global variable `J1` by the other hash function "`d_hash()`". The `CAESAR_1_HASH()` function is more general than "`s_hash()`" since it was extended to 64-bit machines and since it does not assume that `CAESAR_SIZE` is a power of four and that `CAESAR_MODULUS` is a power of two. It also attempts to solve portability issues that occur in "`s_hash()`". Yet, it may be slower than "`s_hash()`".

Note: The result returned by this function may differ accross different machines, since it depends on machine endianness.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_2_HASH (CAESAR_P, CAESAR_SIZE, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the byte string of length `CAESAR_SIZE` pointed to by `CAESAR_P`, and returns this value, which is in the range 0..(`CAESAR_MODULUS` − 1). If either `CAESAR_SIZE` or `CAESAR_MODULUS` is equal to 0, the result is undefined.

Note: This function is derived from the hash function "`d_hash()`" formerly used in the SPIN validation system (see Gerard Holzmann's book "Design and Validation of Computer Protocols", 1991, page 307) with respect to the result returned in global variable `J2`. The `CAESAR_2_HASH()` function is more general than "`d_hash()`" since it was extended to 64-bit machines and since it does not assume

that CAESAR_SIZE is a power of four and that CAESAR_MODULUS is a power of two. It also attempts to solve portability issues that occur in "d_hash()". Yet, it may be slower than "d_hash()".

Note: The result returned by this function may differ accross different machines, since it depends on machine endianness.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_3_HASH (CAESAR_P, CAESAR_SIZE, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the byte string of length CAESAR_SIZE pointed to by CAESAR_P, and returns this value, which is in the range 0..(CAESAR_MODULUS − 1). If either CAESAR_SIZE or CAESAR_MODULUS is equal to 0, the result is undefined.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_4_HASH (CAESAR_P, CAESAR_SIZE, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the byte string of length CAESAR_SIZE pointed to by CAESAR_P, and returns this value, which is in the range 0..(CAESAR_MODULUS − 1). If either CAESAR_SIZE or CAESAR_MODULUS is equal to 0, the result is undefined.

Note: This function is derived from the PJW algorithm given in "Compilers: Principles, Techniques, and Tools" by Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, [pp. 434-438], which was used primarly for compressing character strings. The CAESAR_4_HASH() function removes a limitation of the original PJW algorithm, which always returns a hash value on 24 bits only. If CAESAR_MODULUS is larger than 24 bits, the result of CAESAR_4_HASH() might be also larger than 24 bits.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_5_HASH (CAESAR_P, CAESAR_SIZE, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the byte string of length CAESAR_SIZE pointed to by CAESAR_P, and returns this value, which is in the range 0..(CAESAR_MODULUS − 1). If either CAESAR_SIZE or CAESAR_MODULUS is equal to 0, the result is undefined.

Note: This function performs simple computations based upon XOR operations and bit shifts.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_6_HASH (CAESAR_P, CAESAR_SIZE, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the byte string of length `CAESAR_SIZE` pointed to by `CAESAR_P`, and returns this value, which is in the range 0..(`CAESAR_MODULUS` − 1). If either `CAESAR_SIZE` or `CAESAR_MODULUS` is equal to 0, the result is undefined.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_7_HASH (CAESAR_P, CAESAR_SIZE, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the byte string of length `CAESAR_SIZE` pointed to by `CAESAR_P`, and returns this value, which is in the range 0..(`CAESAR_MODULUS` − 1). If either `CAESAR_SIZE` or `CAESAR_MODULUS` is equal to 0, the result is undefined.

Note: This function is derived from the hash functions proposed by Bob Jenkins, namely "`lookup2()`" for 32-bit machines and "`lookup8()`" for 64-bit machines.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_STATE_0_HASH (CAESAR_S, CAESAR_MODULUS)
   CAESAR_TYPE_STATE CAESAR_S;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the state pointed to by `CAESAR_S` and returns this value, which is in the range 0..(`CAESAR_MODULUS`−1). If `CAESAR_MODULUS` is equal to 0, the result is undefined.

This function is defined by the following equality:

```
   CAESAR_STATE_0_HASH (CAESAR_S, CAESAR_MODULUS) =
   CAESAR_HASH_STATE (CAESAR_S, CAESAR_MODULUS)
```

where `CAESAR_HASH_STATE()` is the hashing function exported by the graph module.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_STATE_1_HASH (CAESAR_S, CAESAR_MODULUS)
   CAESAR_TYPE_STATE CAESAR_S;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
```

```
{ ... }
```

This function computes a hash-value for the state pointed to by `CAESAR_S` and returns this value, which is in the range 0..(`CAESAR_MODULUS`−1). If `CAESAR_MODULUS` is equal to 0, the result is undefined.

This function is defined by the following equality:

```
CAESAR_STATE_1_HASH (CAESAR_S, CAESAR_MODULUS) =
CAESAR_1_HASH (CAESAR_S, CAESAR_HASH_SIZE_STATE (), CAESAR_MODULUS)
```

Additional functions

```
CAESAR_STATE_2_HASH (CAESAR_S, CAESAR_MODULUS)
CAESAR_STATE_3_HASH (CAESAR_S, CAESAR_MODULUS)
CAESAR_STATE_4_HASH (CAESAR_S, CAESAR_MODULUS)
CAESAR_STATE_5_HASH (CAESAR_S, CAESAR_MODULUS)
CAESAR_STATE_6_HASH (CAESAR_S, CAESAR_MODULUS)
CAESAR_STATE_7_HASH (CAESAR_S, CAESAR_MODULUS)
```

are defined in the same way as `CAESAR_STATE_1_HASH()`.

.....

```
CAESAR_TYPE_NATURAL CAESAR_LABEL_0_HASH (CAESAR_L, CAESAR_MODULUS)
   CAESAR_TYPE_LABEL CAESAR_L;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the label pointed to by `CAESAR_L` and returns this value, which is in the range 0..(`CAESAR_MODULUS`−1). If `CAESAR_MODULUS` is equal to 0, the result is undefined.

This function is defined by the following equality:

```
CAESAR_LABEL_0_HASH (CAESAR_L, CAESAR_MODULUS) =
CAESAR_HASH_LABEL (CAESAR_L, CAESAR_MODULUS)
```

where `CAESAR_HASH_LABEL()` is the hashing function exported by the graph module.

.....

```
CAESAR_TYPE_NATURAL CAESAR_LABEL_1_HASH (CAESAR_L, CAESAR_MODULUS)
   CAESAR_TYPE_LABEL CAESAR_L;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the label pointed to by `CAESAR_L` and returns this value, which is in the range 0..(`CAESAR_MODULUS`−1). If `CAESAR_MODULUS` is equal to 0, the result is undefined.

This function is defined by the following equality:

```
CAESAR_LABEL_1_HASH (CAESAR_L, CAESAR_MODULUS) =
CAESAR_1_HASH (CAESAR_L, CAESAR_HASH_SIZE_LABEL (), CAESAR_MODULUS)
```

Additional functions

```
CAESAR_LABEL_2_HASH (CAESAR_L, CAESAR_MODULUS)
CAESAR_LABEL_3_HASH (CAESAR_L, CAESAR_MODULUS)
```

```
CAESAR_LABEL_4_HASH (CAESAR_L, CAESAR_MODULUS)
CAESAR_LABEL_5_HASH (CAESAR_L, CAESAR_MODULUS)
CAESAR_LABEL_6_HASH (CAESAR_L, CAESAR_MODULUS)
CAESAR_LABEL_7_HASH (CAESAR_L, CAESAR_MODULUS)
```

are defined in the same way as `CAESAR_LABEL_1_HASH()`.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_STRING_0_HASH (CAESAR_S, CAESAR_MODULUS)
   CAESAR_TYPE_STRING CAESAR_S;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash-value for the (variable-length, null-terminated) character string pointed to by `CAESAR_S` and returns this value, which is in the range 0..(`CAESAR_MODULUS` − 1). If `CAESAR_MODULUS` is equal to 0, the result is undefined.

. . . . . .

Chapter 9

The "area_1" library (version 1.1)

by Hubert Garavel

9.1 Purpose

The "area_1" library provides primitives for managing "areas", which are memory chunks of various sizes and alignment factors. This library provides for genericity by allowing different objects (states, labels, character strings, user-defined memory chunks) to be handled uniformly.

9.2 Usage

The "area_1" library consists of:

- a predefined header file "`caesar_area_1.h`";
- the precompiled library file "`libcaesar.a`", which implements the features described in "`caesar_area_1.h`".

Note: The "area_1" library is a software layer built above the primitives offered by the "standard" and "hash" libraries, and by the OPEN/CÆSAR graph module.

9.3 Description

An "area" is basically a memory chunk (i.e., a sequence of contiguous bytes) characterized by its (fixed) size and its alignment factor (see the description of `CAESAR_ALIGNMENT_POINTER()` in the "standard" library for a definition of the alignment factor).

There are five different kinds of areas:

- an "ordinary area" may contain any kind of data; the precise contents of an ordinary area is left to the user and is not specified in the context of the "area_1" library;
- an "empty area" is a special area of zero bytes;

- a "state area" is a special area dedicated to contain a state, as described in the graph module, i.e., a value of type `CAESAR_BODY_STATE`;
- a "label area" is a special area dedicated to contain a label, as described in the graph module, i.e., a value of type `CAESAR_BODY_LABEL`;
- a "string area" is a special area dedicated to contain a pointer to a null-terminated character string, i.e., a value of type `CAESAR_TYPE_STRING`. Notice that a string area does not contain a (variable length) character string but a (fixed length) pointer to a (variable length) character string.

9.4 Features

```
typedef CAESAR_TYPE_NATURAL CAESAR_TYPE_AREA_1;
#define CAESAR_EXPONENT_EMPTY_AREA_1     0
#define CAESAR_EXPONENT_STATE_AREA_1     1
#define CAESAR_EXPONENT_LABEL_AREA_1     2
#define CAESAR_EXPONENT_STRING_AREA_1    3
```

Concretely, the type `CAESAR_TYPE_AREA_1` is represented as a numerical type (32-bit or 64-bit natural number depending on the machine architecture). Logically, a value of type `CAESAR_TYPE_AREA_1` is a pair (length field, exponent field), where:

- the exponent field is an unsigned natural number coded on the 4 highest bits (thus, in the range 0..15), and
- the length field is an unsigned natural number coded on the remaining (all but 4) lowest bits, i.e., either coded on 28 bits (and thus, in the range $0...268,435,455$) on 32-bit machines, or coded on 60 bits (and thus, in the range $0...1,152,921,504,606,846,975$) on 64-bit machines.

The different kinds of areas are represented as follows:

- for an ordinary area, the length field is different from zero and represents the size (in bytes) of the area; if the exponent field is equal to a value f different from zero, the alignment factor (in bytes) of the area is equal to 2^{f-1}; the case in which the exponent field is equal to zero corresponds to a backward compatibility situation, which is now obsolete and in which the alignment factor is guessed empirically from the value of the length field (see the definition of `CAESAR_ALIGNMENT_AREA_1()` below);
- for an empty area, the length field is equal to zero and the exponent field is equal to the constant `CAESAR_EXPONENT_EMPTY_AREA_1`;
- for a state area, the length field is equal to zero and the exponent field is equal to the constant `CAESAR_EXPONENT_STATE_AREA_1`;
- for a label area, the length field is equal to zero and the exponent field is equal to the constant `CAESAR_EXPONENT_LABEL_AREA_1`;
- for a string area, the length field is equal to zero and the exponent field is equal to the constant `CAESAR_EXPONENT_STRING_AREA_1`;
- any other case in which the length field is equal to zero is undefined.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_LENGTH_AREA_1 (CAESAR_AREA)
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   { ... }
```

This function returns the length field of the area `CAESAR_AREA`.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_EXPONENT_AREA_1 (CAESAR_AREA)
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   { ... }
```

This function returns the exponent field of the area `CAESAR_AREA`.

. . . . . .

```
CAESAR_AREA_1 CAESAR_AREA_1 (CAESAR_LENGTH, CAESAR_EXPONENT)
   CAESAR_TYPE_NATURAL CAESAR_LENGTH;
   CAESAR_TYPE_NATURAL CAESAR_EXPONENT;
   { ... }
```

This function returns the area with a length field equal to `CAESAR_LENGTH` and an exponent field equal to `CAESAR_EXPONENT`.

Note: this function does not check that its parameters are consistent, e.g., that for an ordinary area the alignment (specified by the exponent field) is an exact divider of the size (specified by the length field).

. . . . . .

```
CAESAR_TYPE_AREA_1 CAESAR_EMPTY_AREA_1 ()
   { ... }
```

This function returns the area with a length field equal to 0 and an exponent field equal to `CAESAR_EXPONENT_EMPTY_AREA_1` (i.e., corresponding to an empty area).

Note: For backward compatibility reasons, `CAESAR_EMPTY_AREA_1()` is equal to 0.

. . . . . .

```
CAESAR_TYPE_AREA_1 CAESAR_STATE_AREA_1 ()
   { ... }
```

This function returns the area with a length field equal to 0 and an exponent field equal to `CAESAR_EXPONENT_STATE_AREA_1` (i.e., corresponding to a state area).

```
CAESAR_TYPE_AREA_1 CAESAR_LABEL_AREA_1 ()
   { ... }
```

This function returns the area with a length field equal to 0 and an exponent field equal to `CAESAR_EXPONENT_LABEL_AREA_1` (i.e., corresponding to a label area).

```
CAESAR_TYPE_AREA_1 CAESAR_STRING_AREA_1 ()
   { ... }
```

This function returns the area with a length field equal to 0 and an exponent field equal to `CAESAR_EXPONENT_STRING_AREA_1` (i.e., corresponding to a string area).

```
CAESAR_TYPE_AREA_1 CAESAR_BYTE_AREA_1 (CAESAR_LENGTH)
   CAESAR_TYPE_NATURAL CAESAR_LENGTH;
   { ... }
```

This function returns the area with a length field equal to `CAESAR_LENGTH` and an exponent field corresponding to a byte or a character (i.e., a memory area with an alignment of 1).

```
CAESAR_TYPE_AREA_1 CAESAR_NATURAL_AREA_1 (CAESAR_LENGTH)
   CAESAR_TYPE_NATURAL CAESAR_LENGTH;
   { ... }
```

This function returns the area with a length field equal to `CAESAR_LENGTH` and an exponent field corresponding to a natural or integer number (i.e., a memory area with an alignment suitable for a value of type `CAESAR_TYPE_NATURAL` or `CAESAR_TYPE_INTEGER`).

Note: In principle, such an area should not contain pointers.

```
CAESAR_TYPE_AREA_1 CAESAR_POINTER_AREA_1 (CAESAR_LENGTH)
   CAESAR_TYPE_NATURAL CAESAR_LENGTH;
   { ... }
```

This function returns the area with a length field equal to `CAESAR_LENGTH` and an exponent field corresponding to a pointer (i.e., a memory area with an alignment equal to `CAESAR_ALIGNMENT_POINTER()`).

Note: In principle, such an area should contain at least one pointer, because pointers usually have

the strongest alignment constraints.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_SIZE_AREA_1 (CAESAR_AREA)
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   { ... }
```

This function returns the size (in bytes) of the area `CAESAR_AREA`:

- for an ordinary area, this size is equal to the length field of `CAESAR_AREA`;
- for an empty area, this size is equal to zero;
- for a state area, this size is given by the `CAESAR_SIZE_STATE()` function exported by the graph module;
- for a label area, this size is given by the `CAESAR_SIZE_LABEL()` function exported by the graph module;
- for a string area, this size is equal to `CAESAR_SIZE_POINTER()` (which corresponds to the size of a character string pointer).

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_HASH_SIZE_AREA_1 (CAESAR_AREA)
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   { ... }
```

This function returns the "hashable" size (in bytes) of the area `CAESAR_AREA`:

- for an ordinary area, this size is equal to the length field of `CAESAR_AREA` (this is an arbitrary definition, since the actual contents of `CAESAR_AREA` are unknown);
- for an empty area, this size is equal to zero (which expresses the fact that an empty area is not appropriate for hashing);
- for a state area, this size is given by the `CAESAR_HASH_SIZE_STATE()` function exported by the graph module;
- for a label area, this size is given by the `CAESAR_HASH_SIZE_LABEL()` function exported by the graph module;
- for a string area, this size is equal to zero (which expresses the fact that a character string pointer, i.e., a value of type `(CAESAR_TYPE_STRING *)`, is not directly appropriate for hashing).

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_ALIGNMENT_AREA_1 (CAESAR_AREA)
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   { ... }
```

This function returns the alignment factor (in bytes) of the area `CAESAR_AREA`:

- for an ordinary area whose exponent field f is different from zero, the alignment factor is equal to 2^{f-1};
- for an ordinary area whose exponent field is equal to zero (backward compatibility case), the alignment factor is guessed empirically from the size of `CAESAR_AREA` (precisely, the alignment factor will be the largest value in the set {1, 2, 4, 8} that divides the size of `CAESAR_AREA` exactly; this is an arbitrary definition, since the actual contents of `CAESAR_AREA` are unknown);
- for an empty area, this alignment factor is equal to one (which expresses the fact that an empty area has no specific alignment constraint);
- for a state area, this alignment factor is given by the `CAESAR_ALIGNMENT_STATE()` function exported by the graph module;
- for a label area, this alignment factor is given by the `CAESAR_ALIGNMENT_LABEL()` function exported by the graph module;
- for a string area, this alignment factor is equal to `CAESAR_ALIGNMENT_POINTER()` (which corresponds to the alignment factor of a character string pointer).

.....

```
void CAESAR_COPY_AREA_1 (CAESAR_P1, CAESAR_P2, CAESAR_SIZE)
   CAESAR_TYPE_POINTER CAESAR_P1;
   CAESAR_TYPE_POINTER CAESAR_P2;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   { ... }
```

This procedure copies the memory area (of `CAESAR_SIZE` bytes) pointed to by `CAESAR_P2` to the location pointed to by `CAESAR_P1`.

Note: This function is implemented as a simple wrapper for the POSIX function `memcpy(3)`.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_COMPARE_EMPTY_AREA_1 (CAESAR_P1, CAESAR_P2)
   CAESAR_TYPE_POINTER CAESAR_P1;
   CAESAR_TYPE_POINTER CAESAR_P2;
   { ... }
```

This function is intended to compare two empty areas and, thus, always returns a value different from 0 (since there is only one empty area).

Note: this function is used by the `CAESAR_USE_COMPARE_FUNCTION_AREA_1()` function described below.

. . . . . .

```
CAESAR_TYPE_BOOLEAN CAESAR_COMPARE_STRING_AREA_1 (CAESAR_S1, CAESAR_S2)
   CAESAR_TYPE_STRING *CAESAR_S1;
   CAESAR_TYPE_STRING *CAESAR_S2;
   { ... }
```

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

Note: This function uses the POSIX function `strcmp(3)`.

Note: this function is used by the `CAESAR_USE_COMPARE_FUNCTION_AREA_1()` function described below.

. . . . . .

```
void CAESAR_USE_COMPARE_FUNCTION_AREA_1 (CAESAR_AREA, CAESAR_COMPARE_FUNCTION)
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   CAESAR_TYPE_COMPARE_FUNCTION *CAESAR_COMPARE_FUNCTION;
   { ... }
```

This procedure modifies the (function pointer) value pointed to by `*CAESAR_COMPARE_FUNCTION` (possibly to assign this function pointer a default value if it is equal to `NULL` before the procedure is invoked) in the following cases:

- if `CAESAR_AREA` is an empty area and if `*CAESAR_COMPARE_FUNCTION` is equal to `NULL`, then the `CAESAR_COMPARE_EMPTY_AREA_1()` function defined above will be assigned to `*CAESAR_COMPARE_FUNCTION`;

  Note: it is not allowed to call this procedure if `CAESAR_AREA` is an empty area, and if `*CAESAR_COMPARE_FUNCTION` is different from both `NULL` and `CAESAR_COMPARE_EMPTY_AREA_1()`, since in this case `CAESAR_COMPARE_EMPTY_AREA_1()` is the only sensible comparison function;

- if `CAESAR_AREA` is a state area and if `*CAESAR_COMPARE_FUNCTION` is equal to `NULL`, then the `CAESAR_COMPARE_STATE()` function of the graph module will be assigned to `*CAESAR_COMPARE_FUNCTION`;

  Note: it is not allowed to call this procedure if `CAESAR_AREA` is a state area, and if `*CAESAR_COMPARE_FUNCTION` is different from both `NULL` and `CAESAR_COMPARE_STATE()`, and if the result of `CAESAR_HASH_SIZE_STATE()` is strictly less than the result of `CAESAR_SIZE_STATE()`, since in this case `CAESAR_COMPARE_STATE()` is the only sensible comparison function;

- if `CAESAR_AREA` is a label area and if `*CAESAR_COMPARE_FUNCTION` is equal to `NULL`, then the `CAESAR_COMPARE_LABEL()` function of the graph module will be assigned to `*CAESAR_COMPARE_FUNCTION`;

  Note: it is not allowed to call this procedure if `CAESAR_AREA` is a label area, and if `*CAESAR_COMPARE_FUNCTION` is different from both `NULL` and `CAESAR_COMPARE_LABEL()`, and if the result of `CAESAR_HASH_SIZE_LABEL()` is strictly less than the result of

CAESAR_SIZE_LABEL(), since in this case CAESAR_COMPARE_LABEL() is the only sensible comparison function;

- if CAESAR_AREA is a string area and if *CAESAR_COMPARE_FUNCTION is equal to NULL, then the CAESAR_COMPARE_STRING_AREA_1() function defined above will be assigned to *CAESAR_COMPARE_FUNCTION.

In any other case, *CAESAR_COMPARE_FUNCTION is kept unchanged.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_COMPARE_AREA_1 (CAESAR_COMPARE_FUNCTION,
                                           CAESAR_P1, CAESAR_P2, CAESAR_SIZE)
   CAESAR_TYPE_COMPARE_FUNCTION CAESAR_COMPARE_FUNCTION;
   CAESAR_TYPE_POINTER CAESAR_P1;
   CAESAR_TYPE_POINTER CAESAR_P2;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   { ... }
```

This function compares the two memory areas (of CAESAR_SIZE bytes) pointed to by CAESAR_P1 and CAESAR_P2 using either CAESAR_COMPARE_FUNCTION if this function pointer is not NULL, or the POSIX function memcmp(3) otherwise. The result is CAESAR_TRUE if and only if both memory areas are found to be equal.

Note: Before calling CAESAR_COMPARE_AREA_1(), the actual value of CAESAR_COMPARE_FUNCTION should have been set by calling the CAESAR_USE_COMPARE_FUNCTION_AREA_1() function.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_HASH_EMPTY_AREA_1 (CAESAR_P, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function is intended to compute an hash-value on empty areas and, thus, always returns 0 (since there is only one empty area). If CAESAR_MODULUS is equal to 0, the result is undefined.

Note: this function is used by the CAESAR_USE_HASH_FUNCTION_AREA_1() function described below.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_HASH_STRING_AREA_1 (CAESAR_P, CAESAR_MODULUS)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function is intended to compute an hash-value on string areas. It returns a value in the range 0..(CAESAR_MODULUS-1), computed from the character string pointed to by *CAESAR_P (and not by CAESAR_P, as CAESAR_P is expected to be of type (CAESAR_TYPE_STRING *), not

CAESAR_TYPE_STRING).

Note: to compute the hash-value, this function invokes the CAESAR_STRING_0_HASH() function of the "hash" library.

Note: this function is used by the CAESAR_USE_HASH_FUNCTION_AREA_1() function described below.

. . . . . .

```
void CAESAR_USE_HASH_FUNCTION_AREA_1 (CAESAR_AREA, CAESAR_HASH_FUNCTION)
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   CAESAR_TYPE_HASH_FUNCTION *CAESAR_HASH_FUNCTION;
   { ... }
```

This procedure modifies the (function pointer) value pointed to by *CAESAR_HASH_FUNCTION (possibly to assign this function pointer a default value if it is equal to NULL before the procedure is invoked) in the following cases:

- if CAESAR_AREA is an empty area and if *CAESAR_HASH_FUNCTION is equal to NULL, then the CAESAR_HASH_EMPTY_AREA_1() function defined above will be assigned to *CAESAR_HASH_FUNCTION;

  Note: it is not allowed to call this procedure if CAESAR_AREA is an empty area, and if *CAESAR_HASH_FUNCTION is different from both NULL and CAESAR_HASH_EMPTY_AREA_1(), since in this case CAESAR_HASH_EMPTY_AREA_1() is the only sensible hashing function;

- if CAESAR_AREA is a state area and if *CAESAR_HASH_FUNCTION is equal to NULL, then the CAESAR_HASH_STATE() function of the graph module will be assigned to *CAESAR_HASH_FUNCTION;

  Note: it is not allowed to call this procedure if CAESAR_AREA is a state area, and if *CAESAR_HASH_FUNCTION is different from both NULL and CAESAR_HASH_STATE(), and if the result of CAESAR_HASH_SIZE_STATE() is equal to zero, since in this case CAESAR_HASH_STATE() is the only sensible hashing function;

- if CAESAR_AREA is a label area and if *CAESAR_HASH_FUNCTION is equal to NULL, then the CAESAR_HASH_LABEL() function of the graph module will be assigned to *CAESAR_HASH_FUNCTION;

  Note: it is not allowed to call this procedure if CAESAR_AREA is a label area, and if *CAESAR_HASH_FUNCTION is different from both NULL and CAESAR_HASH_LABEL(), and if the result of CAESAR_HASH_SIZE_LABEL() is equal to zero, since in this case CAESAR_HASH_LABEL() is the only sensible hashing function;

- if CAESAR_AREA is a string area and if *CAESAR_HASH_FUNCTION is equal to NULL, then the CAESAR_HASH_STRING_AREA_1() function of the "hash" library will be assigned to *CAESAR_HASH_FUNCTION.

In any other case, *CAESAR_HASH_FUNCTION is kept unchanged.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_HASH_AREA_1 (CAESAR_HASH_FUNCTION, CAESAR_P,
                                        CAESAR_HASH_SIZE, CAESAR_MODULUS)
   CAESAR_TYPE_HASH_FUNCTION CAESAR_HASH_FUNCTION;
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_HASH_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_MODULUS;
   { ... }
```

This function computes a hash value for the memory area pointed to by `CAESAR_P` using either `CAESAR_HASH_FUNCTION` if this function pointer is not `NULL`, or function `CAESAR_0_HASH()` of the "hash" library otherwise. Hashing is performed on the `CAESAR_HASH_SIZE` first bytes and the hash value returned is in the range 0..(`CAESAR_MODULUS` − 1).

Note: Before calling `CAESAR_HASH_AREA_1()`, the actual value of `CAESAR_HASH_FUNCTION` should have been set by calling the `CAESAR_USE_HASH_FUNCTION_AREA_1()` function.

. . . . . .

```
CAESAR_TYPE_STRING CAESAR_CONVERT_EMPTY_AREA_1 (CAESAR_P)
   CAESAR_TYPE_POINTER CAESAR_P;
   { ... }
```

This function returns a constant, user-readable character string representing the empty area.

Note: It is not allowed to modify the character string returned by `CAESAR_CONVERT_EMPTY_AREA_1()` nor to free it, for instance using `free(3)`.

Note: this function is used by the `CAESAR_USE_CONVERT_FUNCTION_AREA_1()` function described below.

. . . . . .

```
CAESAR_TYPE_STRING CAESAR_CONVERT_STRING_AREA_1 (CAESAR_S)
   CAESAR_TYPE_STRING *CAESAR_S;
   { ... }
```

This function returns the character string pointed to by `*CAESAR_S`.

Note: this function is used by the `CAESAR_USE_CONVERT_FUNCTION_AREA_1()` function described below.

. . . . . .

```
CAESAR_TYPE_STRING CAESAR_CONVERT_BINARY_AREA_1 (CAESAR_P, CAESAR_SIZE)
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   { ... }
```

This function returns a pointer to a character string corresponding to the hexadecimal representation for the `CAESAR_SIZE` bytes-long memory chunk pointed to by `CAESAR_P`.

Note: It is not allowed to modify the character string returned by `CAESAR_CONVERT_BINARY_AREA_1()`

nor to free it, for instance using `free(3)`.

Note: this function is used by the `CAESAR_CONVERT_AREA_1()` function described below.

.....

```
void CAESAR_USE_CONVERT_FUNCTION_AREA_1 (CAESAR_AREA, CAESAR_CONVERT_FUNCTION)
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   CAESAR_TYPE_CONVERT_FUNCTION *CAESAR_CONVERT_FUNCTION;
   { ... }
```

This procedure modifies the (function pointer) value pointed to by `*CAESAR_CONVERT_FUNCTION` (possibly to assign this function pointer a default value if it is equal to `NULL` before the procedure is invoked) in the following cases:

- if `CAESAR_AREA` is an empty area and if `*CAESAR_CONVERT_FUNCTION` is equal to `NULL`, then the `CAESAR_CONVERT_EMPTY_AREA_1()` function defined above will be assigned to `*CAESAR_CONVERT_FUNCTION`;
- if `CAESAR_AREA` is a state area and if `*CAESAR_CONVERT_FUNCTION` is equal to `NULL`, then the effect of this procedure is undefined;
- if `CAESAR_AREA` is a label area and if `*CAESAR_CONVERT_FUNCTION` is equal to `NULL`, then the `CAESAR_STRING_LABEL()` function of the graph module will be assigned to `*CAESAR_CONVERT_FUNCTION`;
- if `CAESAR_AREA` is a string area and if `*CAESAR_CONVERT_FUNCTION` is equal to `NULL`, then the `CAESAR_CONVERT_STRING_AREA_1()` function defined above will be assigned to `*CAESAR_CONVERT_FUNCTION`.

In any other case, `*CAESAR_CONVERT_FUNCTION` is kept unchanged.

.....

```
CAESAR_TYPE_STRING CAESAR_CONVERT_AREA_1 (CAESAR_CONVERT_FUNCTION, CAESAR_P,
                                          CAESAR_SIZE)
   CAESAR_TYPE_CONVERT_FUNCTION CAESAR_CONVERT_FUNCTION;
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   { ... }
```

This function converts the memory area (of `CAESAR_SIZE` bytes) pointed to by `CAESAR_P` into a character string using either `CAESAR_CONVERT_FUNCTION` if this function pointer is not `NULL`, or function `CAESAR_CONVERT_BINARY_AREA_1()` otherwise.

Note: Before calling `CAESAR_CONVERT_AREA_1()`, the actual value of `CAESAR_CONVERT_FUNCTION` should have been set by calling the `CAESAR_USE_CONVERT_FUNCTION_AREA_1()` function.

.....

```
void CAESAR_PRINT_EMPTY_AREA_1 (CAESAR_FILE, CAESAR_P)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_POINTER CAESAR_P;
   { ... }
```

This procedure prints to file `CAESAR_FILE` a constant, user-readable character string representing the empty area.

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

Note: this function is used by the `CAESAR_USE_PRINT_FUNCTION_AREA_1()` function described below.

.....

```
void CAESAR_PRINT_STRING_AREA_1 (CAESAR_FILE, CAESAR_S)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_STRING *CAESAR_S;
   { ... }
```

This procedure prints to file `CAESAR_FILE` the character string pointed to by `*CAESAR_S`.

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

Note: this function is used by the `CAESAR_USE_PRINT_FUNCTION_AREA_1()` function described below.

.....

```
void CAESAR_PRINT_BINARY_AREA_1 (CAESAR_FILE, CAESAR_P, CAESAR_SIZE)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   { ... }
```

This procedure prints to file `CAESAR_FILE` the character string corresponding to the hexadecimal representation for the `CAESAR_SIZE` bytes-long memory chunk pointed to by `CAESAR_P`.

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

Note: this function is used by the `CAESAR_PRINT_AREA_1()` function described below.

.....

```
void CAESAR_USE_PRINT_FUNCTION_AREA_1 (CAESAR_AREA, CAESAR_PRINT_FUNCTION)
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   CAESAR_TYPE_PRINT_FUNCTION *CAESAR_PRINT_FUNCTION;
   { ... }
```

This procedure modifies the (function pointer) value pointed to by `*CAESAR_PRINT_FUNCTION` (possibly to assign this function pointer a default value if it is equal to `NULL` before the procedure is invoked) in the following cases:

- if `CAESAR_AREA` is an empty area and if `*CAESAR_PRINT_FUNCTION` is equal to `NULL`, then the `CAESAR_PRINT_EMPTY_AREA_1()` function defined above will be assigned to `*CAESAR_PRINT_FUNCTION`;
- if `CAESAR_AREA` is a state area and if `*CAESAR_PRINT_FUNCTION` is equal to `NULL`, then the `CAESAR_PRINT_STATE()` function of the graph module will be assigned to `*CAESAR_PRINT_FUNCTION`;
- if `CAESAR_AREA` is a label area and if `*CAESAR_PRINT_FUNCTION` is equal to `NULL`, then the `CAESAR_PRINT_LABEL()` function of the graph module will be assigned to `*CAESAR_PRINT_FUNCTION`;
- if `CAESAR_AREA` is a string area and if `*CAESAR_PRINT_FUNCTION` is equal to `NULL`, then the `CAESAR_PRINT_STRING_AREA_1()` function defined above will be assigned to `*CAESAR_PRINT_FUNCTION`.

In any other case, `*CAESAR_PRINT_FUNCTION` is kept unchanged.

.....

```
void CAESAR_PRINT_AREA_1 (CAESAR_PRINT_FUNCTION, CAESAR_FILE, CAESAR_P,
                          CAESAR_SIZE)
   CAESAR_TYPE_PRINT_FUNCTION CAESAR_PRINT_FUNCTION;
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_POINTER CAESAR_P;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   { ... }
```

This function prints to file `CAESAR_FILE` a character string representing the memory area (of `CAESAR_SIZE` bytes) pointed to by `CAESAR_P`. Printing is done using either `CAESAR_PRINT_FUNCTION` if this function pointer is not `NULL`, or function `CAESAR_PRINT_BINARY_AREA_1()` otherwise.

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

Note: Before calling `CAESAR_PRINT_AREA_1()`, the actual value of `CAESAR_PRINT_FUNCTION` should have been set by calling the `CAESAR_USE_PRINT_FUNCTION_AREA_1()` function.

.....

Chapter 10

The "bitmap" library (version 1.5)

by Hubert Garavel

10.1 Purpose

The "bitmap" library provides primitives for implementing the "bit state space" verification technique proposed by Gerard Holzmann.

10.2 Usage

The "bitmap" library consists of:

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

Note: The "bitmap" library is a software layer built above the primitives offered by the "standard" and "hash" libraries.

10.3 Description

A "bitmap" of size N is basically an array of N bits numbered from 0 to $N-1$. The value of N is usually large (e.g., some tenth million states).

Additionally, statistics are attached to each bitmap. These statistics consist of a "success counter" (which counts how many bits equal to 1 have been read) and a "failure counter" (which counts how many bits equal to 0 have been read).

10.4 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_BITMAP;
```

This type denotes a pointer to the concrete representation of a bitmap, which is supposed to be "opaque".

. . . . . .

```
void CAESAR_CREATE_BITMAP (CAESAR_B, CAESAR_SIZE, CAESAR_PRIME)
   CAESAR_TYPE_BITMAP *CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_SIZE;
   CAESAR_TYPE_BOOLEAN CAESAR_PRIME;
   { ... }
```

This procedure allocates a bitmap using `CAESAR_CREATE()` and assigns its address to `*CAESAR_B`. The size N of this bitmap is determined by the values of formal parameters `CAESAR_SIZE` and `CAESAR_PRIME`, and also by the amount of memory available.

Note: because `CAESAR_TYPE_BITMAP` is a pointer type, any variable `CAESAR_B` of type `CAESAR_TYPE_BITMAP` must be allocated before used, for instance using:

```
CAESAR_CREATE_BITMAP (&CAESAR_B, ...)
```

If the value of `CAESAR_SIZE` is different from 0, then the number N of bits in the bitmap will be `CAESAR_SIZE`.

If the value of `CAESAR_SIZE` is equal to 0, then N will be given a default value as large as possible.

Note: in this case, the bitmap will fill most of the memory space available for the current process. Therefore, if `CAESAR_CREATE_BITMAP()` is to be called with `CAESAR_SIZE` = 0, it should be called only after having allocated all the other data structures (e.g., stacks, ...), otherwise there may be not enough memory for these data structures.

In both cases, the value of N can be reduced to a smaller value as to fit into the amount of available memory.

If the value of `CAESAR_PRIME` is different from 0, the value of N can also be reduced to the immediately smaller prime number (since some hash functions require prime modulus). If the value of `CAESAR_PRIME` is equal to 0, the value of N is not changed.

If (in spite of various attempts) the allocation fails, the `NULL` value is assigned to `*CAESAR_B`.

If the allocation succeeds, the final value of N can be known using the function `CAESAR_SIZE_BITMAP()` (see below).

If the allocation succeeds, the N bits of the bitmap are initialized to 0. The success and failure counters attached to the bitmap are also initialized to 0.

Note: since variable `CAESAR_SIZE` is a value of type `CAESAR_TYPE_NATURAL`, a bitmap can contain at most $2^{8n} - 1$ bits, where:

$$n = \texttt{sizeof (CAESAR\_TYPE\_NATURAL)}$$

It is assumed that $n \geq 4$. For $n = 4$, this makes 4,294,967,295 bits, that is approximately 537 Megabytes of memory, which is currently enough.

. . . . . .

```
void CAESAR_DELETE_BITMAP (CAESAR_B)
```

```
   CAESAR_TYPE_BITMAP *CAESAR_B;
   { ... }
```

This procedure frees the memory space corresponding to the bitmap pointed to by `*CAESAR_B` using `CAESAR_DELETE()`. Afterwards, the `NULL` value is assigned to `*CAESAR_B`.

.....

```
void CAESAR_PURGE_BITMAP (CAESAR_B)
   CAESAR_TYPE_BITMAP CAESAR_B;
   { ... }
```

This procedure empties the bitmap pointed to by `CAESAR_B` without deleting it. Afterwards, this bitmap is exactly in the same state as after its creation using `CAESAR_CREATE_BITMAP()`. Its size remains unchanged.

.....

```
CAESAR_TYPE_NATURAL CAESAR_SIZE_BITMAP (CAESAR_B)
   CAESAR_TYPE_BITMAP CAESAR_B;
   { ... }
```

This function returns the size (i.e., number of bits) of the bitmap pointed to by `CAESAR_B`. This size is determined when the bitmap is created using `CAESAR_CREATE_BITMAP()` and remains constant.

.....

```
void CAESAR_SET_BITMAP (CAESAR_B, CAESAR_I)
   CAESAR_TYPE_BITMAP CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   { ... }
```

This procedure sets to 1 the `CAESAR_I`$^{\text{th}}$ bit of the bitmap pointed to by `CAESAR_B`. The value of `CAESAR_I` is such that:

$$0 \leq \texttt{CAESAR\_I} \leq \texttt{CAESAR\_SIZE\_BITMAP (CAESAR\_B)}$$

It is usually the result of some hash-code computation.

.....

```
void CAESAR_RESET_BITMAP (CAESAR_B, CAESAR_I)
   CAESAR_TYPE_BITMAP CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   { ... }
```

This procedure sets to 0 the `CAESAR_I`$^{\text{th}}$ bit of the bitmap pointed to by `CAESAR_B`. The value of `CAESAR_I` is such that:

$$0 \leq \texttt{CAESAR\_I} \leq \texttt{CAESAR\_SIZE\_BITMAP (CAESAR\_B)}$$

It is usually the result of some hash-code computation.

. . . . . .

```
CAESAR_TYPE_BOOLEAN CAESAR_TEST_BITMAP (CAESAR_B, CAESAR_I)
   CAESAR_TYPE_BITMAP CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   { ... }
```

This function returns 0 if the `CAESAR_I`$^{\text{th}}$ bit of the bitmap pointed to by `CAESAR_B` is equal to 0, or a value different from 0 if this bit is equal to 1. The value of `CAESAR_I` is such that:

$$0 \leq \texttt{CAESAR\_I} \leq \texttt{CAESAR\_SIZE\_BITMAP (CAESAR\_B)}$$

It is usually the result of some hash-code computation.

A return value of 0 increments the failure counter attached to the bitmap pointed to by `CAESAR_B`, whereas a return value of 1 increments the success counter.

. . . . . .

```
CAESAR_TYPE_BOOLEAN CAESAR_TEST_AND_SET_BITMAP (CAESAR_B, CAESAR_I)
   CAESAR_TYPE_BITMAP CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   { ... }
```

This function returns 0 if the `CAESAR_I`$^{\text{th}}$ bit of the bitmap pointed to by `CAESAR_B` is equal to 0, or a value different from 0 if this bit is equal to 1. The value of `CAESAR_I` is such that:

$$0 \leq \texttt{CAESAR\_I} \leq \texttt{CAESAR\_SIZE\_BITMAP (CAESAR\_B)}$$

It is usually the result of some hash-code computation.

The `CAESAR_I`$^{\text{th}}$ bit of the bitmap pointed to by `CAESAR_B` is set to 1 if it was equal to 0.

A return value of 0 increments the failure counter attached to the bitmap pointed to by `CAESAR_B`, whereas a return value of 1 increments the success counter.

. . . . . .

```
CAESAR_TYPE_BOOLEAN CAESAR_TEST_AND_RESET_BITMAP (CAESAR_B, CAESAR_I)
   CAESAR_TYPE_BITMAP CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   { ... }
```

This function returns 0 if the `CAESAR_I`$^{\text{th}}$ bit of the bitmap pointed to by `CAESAR_B` is equal to 0, or a value different from 0 if this bit is equal to 1. The value of `CAESAR_I` is such that:

$$0 \leq \texttt{CAESAR\_I} \leq \texttt{CAESAR\_SIZE\_BITMAP (CAESAR\_B)}$$

It is usually the result of some hash-code computation.

The `CAESAR_I`th bit of the bitmap pointed to by `CAESAR_B` is set to 0 if it was equal to 1.

A return value of 0 increments the failure counter attached to the bitmap pointed to by `CAESAR_B`, whereas a return value of 1 increments the success counter.

.....

```
CAESAR_TYPE_NATURAL CAESAR_ZERO_BITMAP (CAESAR_B)
   CAESAR_TYPE_BITMAP CAESAR_B;
   { ... }
```

This function returns the number of bits which are equal to 0 in the bitmap pointed to by `CAESAR_B`.

.....

```
CAESAR_TYPE_NATURAL CAESAR_ONE_BITMAP (CAESAR_B)
   CAESAR_TYPE_BITMAP CAESAR_B;
   { ... }
```

This function returns the number of bits which are equal to 1 in the bitmap pointed to by `CAESAR_B`.

Note: for any bitmap `CAESAR_B`:

`CAESAR_ZERO_BITMAP (CAESAR_B)` + `CAESAR_ONE_BITMAP (CAESAR_B)`

is equal to:

`CAESAR_SIZE_BITMAP (CAESAR_B)`

.....

```
CAESAR_TYPE_NATURAL CAESAR_FAILURE_BITMAP (CAESAR_B)
   CAESAR_TYPE_BITMAP CAESAR_B;
   { ... }
```

This function returns the value of the failure counter of the bitmap pointed to by `CAESAR_B`, i.e., the number of searches that failed (see functions `CAESAR_TEST_BITMAP()`, `CAESAR_TEST_AND_SET_BITMAP()`, and `CAESAR_TEST_AND_RESET()` above).

.....

```
CAESAR_TYPE_NATURAL CAESAR_SUCCESS_BITMAP (CAESAR_B)
   CAESAR_TYPE_BITMAP CAESAR_B;
   { ... }
```

This function returns the value of the success counter of the bitmap pointed to by `CAESAR_B`, i.e., the number of searches that succeeded (see functions `CAESAR_TEST_BITMAP()`, `CAESAR_TEST_AND_SET_BITMAP()`, and `CAESAR_TEST_AND_RESET()` above).

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_BITMAP (CAESAR_B, CAESAR_FORMAT)
   CAESAR_TYPE_BITMAP CAESAR_B;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the bitmap pointed to by `CAESAR_B` will be printed by the procedure `CAESAR_PRINT_BITMAP()` (see below). Currently, the following formats are available:

- With format 0, statistical information about the bitmap is displayed such as: the size in bytes, the number of bits, the number of bits equal to 0, the number of bits equal to 1, the success counter, the failure counter, etc.
- With format 1, the contents of the bitmap are printed in hexadecimal. This can be useful for debugging bitmaps of small size.
- With format 2, the list of bits which are equal to 0 is printed. This can be useful for debugging bitmaps with almost all bits equal to 1.
- With format 3, the list of bits which are equal to 1 is printed. This can be useful for debugging bitmaps with almost all bits equal to 0.
- (no other format available yet).

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

When called with `CAESAR_FORMAT` between 0 and 3, this fonction sets the current format of `CAESAR_B` 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_B` 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_B`. If `CAESAR_FORMAT` is equal to the constant `CAESAR_MAXIMAL_FORMAT`, the result is the maximal format value (i.e., 3). In all other cases, the effect of this function is undefined.

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_BITMAP ()
   { ... }
```

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

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

```
void CAESAR_PRINT_BITMAP (CAESAR_FILE, CAESAR_B)
```

```
CAESAR_TYPE_FILE CAESAR_FILE;
CAESAR_TYPE_BITMAP CAESAR_B;
{ ... }
```

This procedure prints to file `CAESAR_FILE` a character string containing information about the bitmap pointed to by `CAESAR_B`. The nature of the information is determined by the current format of the bitmap pointed to by `CAESAR_B`.

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

. . . . . .

Chapter 11

The "table_1" library (version 1.1)

by Hubert Garavel

11.1 Purpose

The "table_1" library provides primitives for managing a "state space". It can be used either for breadth-first or depth-first search in the state graph.

11.2 Usage

The "table_1" library consists of:

- a predefined header file "`caesar_table_1.h`";
- the precompiled library file "`libcaesar.a`", which implements the features described in "`caesar_table_1.h`".

Note: The "table_1" library is a software layer built above the primitives offered by the "standard", "area_1", and "hash" libraries, and by the OPEN/CÆSAR graph module.

11.3 Description

A "table" is basically a set of items.

Each item in the table is basically a byte string of fixed size. All items in a given table have the same size. An item can be considered as a tuple with two fields, whose size and contents are freely determined by the user:

- (1) a "base" field, that is a byte string of fixed size. In a given table, all base fields have the same size. This size must be greater than zero.

  More often than not, the base field contains a state (as defined in the graph module). However, this is not mandatory, and base fields can contain other information than states.

- (2) a "mark" field, that is a byte string whose size and contents are freely determined by the user. In a given table, all mark fields have the same size, which must be greater or equal to zero. Pointers to mark fields will be considered as values of type `CAESAR_TYPE_POINTER`; "mark" fields are always aligned on appropriate boundaries so that the user can put any information in these fields without alignment problem.

The user also determines the nature of the data stored in these fields, which is not meaningful to the "table_1" library.

Invariant property 1: the table is organized in such a way that all items have different base fields. Said differently, two items in a given table cannot have identical base fields (but they can have identical mark fields).

Invariant property 2: the number of items in a given table never decreases. New items can be inserted in the table, existing items can be replaced by new ones, but no item can be removed if it is not replaced by another one. Exception to this rule: it is possible to purge the table, i.e., to remove simultaneously all its items.

Invariant property 3: it is not allowed to modify the base field of any item in the table (except if this item is to be replaced by another one that has exactly the same hash value, which is unlikely). But it is possible to modify the mark field of any item.

Each item in a table is given a unique identification number ("index") which is a natural number. A table can contain no more than a maximum of M items, with indexes between 0 and $M - 1$. Currently, $M = 2^{29} = 536,870,912$ on 32-bit machines and $M = 2^{34} = 17,179,869,184$ on 64-bit machines. But, for each table, the user can also limit the maximal number of items to a lesser bound $N \leq M$.

When the table overflows (either because the maximum number of items is reached or because there is no enough memory to store new items), an action chosen by the user (e.g., abort, recovery, etc.) is performed.

Each item in the table can be accessed in three different ways:

- (1) by using its address (i.e., a pointer to the memory location where it is stored in the table),
- (2) by using its index,
- (3) by using its base field.

The table data structure establishes a correspondence between these three data. Indeed:

- given an address, one can retrieve the index, the base field, and the mark field of the corresponding item;
- given an index, one can retrieve the address, the base field, and the mark field of the corresponding item;
- given a base field, one can retrieve the address, the index, and the mark field of the corresponding item.

Retrieving the address and the index of an item from its base field involves some associative search. To allow fast retrievals, an hash-table is associated to each table. This is quite transparent from the user's point of view. Only the base field is taken into account when computing the hash-value and comparing items; the mark field is not meaningful for the search.

Two special variables are associated to a given table:

- the "put index" corresponds to the last item inserted in the table. Initialized to zero, the put index is incremented each time a new item is inserted. Therefore, the first item is numbered 0, the second one is numbered 1, etc.

  The "put index" is always useful, whatever the way the graph is explored: breadth-first, depth-first, etc.

- the "get index" is associated to the last item consulted in the table. Initialized to zero, the get index is incremented each time a new item is consulted. Since only the items previously inserted in the table can be consulted, the get index is always less or equal to the put index.

  The "get index" can be used to consult sequentially all the items, in the same order as they have been inserted in the table. Therefore it can be used for (pseudo) breadth-first exploration, but not for depth-first exploration.

Additionally, statistics are attached to each table. These statistics consist of a "success counter" and a "failure counter", which respectively count how many retrievals (given the base field) have succeeded and failed.

11.4 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_TABLE_1;
```

This type denotes a pointer to the concrete representation of a table. The table representation is supposed to be "opaque".

.....

```
typedef CAESAR_TYPE_NATURAL CAESAR_TYPE_INDEX_TABLE_1;
```

This type denotes an index, which is a natural number.

.....

```
#define CAESAR_NULL_INDEX_TABLE_1 ((CAESAR_TYPE_INDEX_TABLE_1) -1L)
```

This constant denotes a special index value corresponding to the largest unsigned integer. Since item indexes are always in the range $0..M-1$, no item index can be equal to `CAESAR_NULL_INDEX_TABLE_1`.

.....

```
CAESAR_TYPE_INDEX_TABLE_1 CAESAR_MAX_INDEX_TABLE_1 ()
   { ... }
```

This function returns the value of M, i.e., the maximal number of items that can be stored in a table. Since item indexes are always in the range $0..M-1$, no item index can be equal to `CAESAR_MAX_INDEX_TABLE_1()`.

```
typedef void (*CAESAR_TYPE_OVERFLOW_FUNCTION_TABLE_1) (CAESAR_TYPE_TABLE_1);
```

`CAESAR_TYPE_OVERFLOW_FUNCTION_TABLE_1` is the "pointer to an overflow procedure" type used in the "table_1" library. An overflow procedure takes one parameter of type `CAESAR_TYPE_TABLE_1`. Examples of overflow procedures are `CAESAR_OVERFLOW_SIGNAL_TABLE_1()`, `CAESAR_OVERFLOW_ABORT_TABLE_1()`, and `CAESAR_OVERFLOW_IGNORE_TABLE_1()` defined below.

```
void CAESAR_OVERFLOW_SIGNAL_TABLE_1 (CAESAR_T)
    CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure is a possible action that can be performed in case the table pointed to by `CAESAR_T` overflows (either because the maximum number of items for `CAESAR_T` is reached or because there is no enough memory to store new items).

It first prints an error message to the standard output, and also various statistics about the table. Then, it returns. Practically, if the table is used for state space exploration, this means that some portions of the graph will not be explored, but an error message will be issued.

```
void CAESAR_OVERFLOW_ABORT_TABLE_1 (CAESAR_T)
    CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure is a possible action that can be performed in case the table pointed to by `CAESAR_T` overflows (either because the maximum number of items for `CAESAR_T` is reached or because there is no enough memory to store new items).

It first prints an error message to the standard output, and also various statistics about the table. Then, it aborts the program using the C function `exit(3)`. The error code 1 is returned.

```
void CAESAR_OVERFLOW_IGNORE_TABLE_1 (CAESAR_T)
    CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure is a possible action that can be performed in case the table pointed to by `CAESAR_T` overflows (either because the maximum number of items for `CAESAR_T` is reached or because there is no enough memory to store new items).

It does nothing and returns. Practically, if the table is used for state space exploration, this means that some portions of the graph will not be explored; they are silently ignored.

```
void CAESAR_CREATE_TABLE_1 (CAESAR_T, CAESAR_BASE_AREA, CAESAR_MARK_AREA,
                            CAESAR_LIMIT_SIZE, CAESAR_HASH_SIZE,
                            CAESAR_PRIME, CAESAR_COMPARE, CAESAR_HASH,
                            CAESAR_PRINT, CAESAR_OVERFLOW)
   CAESAR_TYPE_TABLE_1 *CAESAR_T;
   CAESAR_TYPE_AREA_1 CAESAR_BASE_AREA;
   CAESAR_TYPE_AREA_1 CAESAR_MARK_AREA;
   CAESAR_TYPE_NATURAL CAESAR_LIMIT_SIZE;
   CAESAR_TYPE_NATURAL CAESAR_HASH_SIZE;
   CAESAR_TYPE_BOOLEAN CAESAR_PRIME;
   CAESAR_TYPE_COMPARE_FUNCTION CAESAR_COMPARE;
   CAESAR_TYPE_HASH_FUNCTION CAESAR_HASH;
   CAESAR_TYPE_PRINT_FUNCTION CAESAR_PRINT;
   CAESAR_TYPE_OVERFLOW_FUNCTION_TABLE_1 CAESAR_OVERFLOW;
   { ... }
```

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

Note: when it is called, this procedure does not allocate at once all the memory needed to represent the table: the table will grow progressively as new items are inserted. Consequently, if `CAESAR_CREATE_TABLE_1()` returns a value different from `NULL`, this does not mean that no overflow will occur in the future.

Note: because `CAESAR_TYPE_TABLE_1` is a pointer type, any variable `CAESAR_T` of type `CAESAR_TYPE_TABLE_1` must be allocated before used, for instance using:

```
CAESAR_CREATE_TABLE_1 (&CAESAR_T, ...);
```

The value of `CAESAR_BASE_AREA` determines the (constant) size and (constant) alignment factor of the base field in the table. In the particular case where base fields are used to store states (resp. labels, strings), one must give the value `CAESAR_STATE_AREA_1()` (resp. `CAESAR_LABEL_AREA_1()`, `CAESAR_STRING_AREA_1()`) to the formal parameter `CAESAR_BASE_AREA`.

Note: For backward compatibility reasons, if `CAESAR_BASE_AREA` is equal to `CAESAR_EMPTY_AREA_1()`, it will be treated exactly like `CAESAR_STATE_AREA_1()`, i.e., specifying an empty area for the base field is equivalent to specifying a state area. However, relying on this feature is not recommended and this case will no longer be considered in the sequel of this manual.

The value of `CAESAR_MARK_AREA` determines the (constant) size and (constant) alignment factor of the mark field according to the specifications of the "area_1" library. In particular, if `CAESAR_MARK_AREA` is equal to `CAESAR_EMPTY_AREA_1()`, there will be no mark field in the table.

Each item in the table will be represented as a byte string of fixed size `caesar_item_size`, such that `caesar_item_size` is strictly greater than `caesar_base_size + caesar_mark_size`, where `caesar_base_size` denotes the size (in bytes) of the base field (i.e., `CAESAR_SIZE_AREA_1 (CAESAR_BASE_AREA)`) and where `caesar_mark_size` denotes the size (in bytes) of the mark field, if any (i.e., `CAESAR_SIZE_AREA_1 (CAESAR_MARK_AREA)`).

An item in the table contains not only the base field and the mark field, but also additional information needed for internal management. Also, "padding" bytes may be inserted around the base and mark fields to ensure that these fields are

correctly aligned according to `CAESAR_ALIGNMENT_AREA_1 (CAESAR_BASE_AREA)` and `CAESAR_ALIGNMENT_AREA_1 (CAESAR_MARK_AREA)`.

The value of `CAESAR_LIMIT_SIZE` determines the maximal number of items that can be stored in the table; all indexes will consequently be in the range 0..`CAESAR_LIMIT_SIZE` − 1. It must be less or equal to M. If it is equal to zero, it is replaced by the default value M.

Note: in order to spare memory, the value of `CAESAR_LIMIT_SIZE` (which is an upper bound on the number of items to be inserted in the table) should be as small as possible. This can only be done if the user has some knowledge about the way the table will be used. The value of `CAESAR_LIMIT_SIZE` (or its default value if `CAESAR_LIMIT_SIZE` is equal to zero) might be reduced automatically if it exceeds the number of all possible different base fields or if it exceeds the amount of physical memory available (which is either specified by the environment variable `$CADP_MEMORY` or determined automatically by the "`cadp_memory`" program).

The value of `CAESAR_HASH_SIZE` determines the number of entries in the hash-table associated to the table. If `CAESAR_HASH_SIZE` is different from zero, it remains constant during the entire existence of the table (static hashing scheme). If `CAESAR_HASH_SIZE` is equal to zero, it is replaced with a default value greater than zero that might increase automatically when a sufficiently large number of items have been inserted into the table (dynamic hashing scheme).

Note: in order to spare memory, the value of `CAESAR_HASH_SIZE` (or its default value if `CAESAR_HASH_SIZE` is equal to zero) might be reduced automatically if it exceeds the value of `CAESAR_LIMIT_SIZE`, i.e., the maximal number of items that can be stored in the table, or if it exceeds the maximal number of different hash values that can be obtained taking into account the "hashable" size of `CAESAR_BASE_AREA`.

If the value of `CAESAR_PRIME` is equal to `CAESAR_TRUE` and if the value of `CAESAR_HASH_SIZE` is not a prime number, this value will be replaced by the nearest smaller prime number (since some hash functions require prime modulus). Otherwise, the value of `CAESAR_HASH_SIZE` will be kept unchanged.

The actual value of the formal parameter `CAESAR_COMPARE` will be stored and associated to the table pointed to by `*CAESAR_T`. It will be used as a comparison function when it is necessary to decide whether two base fields are equal or not.

Precisely, the actual value of `CAESAR_COMPARE` should be a pointer to a comparison function with two parameters `caesar_base_1` and `caesar_base_2` that returns `CAESAR_TRUE` if the two base fields pointed to by `caesar_base_1` and `caesar_base_2` are equal.

If the actual value of the formal parameter `CAESAR_COMPARE` is `NULL`, it is replaced by a pointer to a default comparison function that depends on the value of `CAESAR_BASE_AREA` and is determined according to the rules specified for function `CAESAR_USE_COMPARE_FUNCTION_AREA_1()` of the "area_1" library.

The actual value of the formal parameter `CAESAR_HASH` will be stored and associated to the table pointed to by `*CAESAR_T`. It will be used as a hash-function when it is necessary to compute a hash-value for searching or inserting an item in the table.

Precisely, the actual value of `CAESAR_HASH` should be a pointer to a hash function with two parameters `caesar_pointer` and `caesar_modulus` that returns a natural number in the range 0..`caesar_modulus` − 1.

If the actual value of the formal parameter `CAESAR_HASH` is `NULL`, it is replaced by a pointer to a default hash function that depends on the value of `CAESAR_BASE_AREA` and is determined according to the rules specified for function `CAESAR_USE_HASH_FUNCTION_AREA_1()` of the "area_1" library.

Note: for backward compatibility reasons, the current implementation of `CAESAR_CREATE_TABLE_1()`

tries to handle the case where CAESAR_HASH points to an hash function with three parameters (such as the CAESAR_0_HASH(), CAESAR_1_HASH(), ... functions provided by the "hash" library) instead of two. However, this support for hash functions with three parameters only occurs under very specific circumstances (e.g., if CAESAR_BASE_AREA has a null exponent field and a non-null length field). Relying on this feature is not recommended and this case will no longer be considered in the sequel of this manual.

The actual value of the formal parameter CAESAR_PRINT will be stored and associated to the table pointed to by *CAESAR_T. It will be used subsequently to print the items of this table.

Precisely, the actual value of CAESAR_PRINT should be a pointer to a printing procedure with two parameters caesar_file and caesar_item that prints to file caesar_file information about the contents (base field and/or mark field, if any) of the item pointed to by caesar_item.

If the actual value of the formal parameter CAESAR_PRINT is NULL, it is replaced by a pointer to a default procedure that prints the base field and the mark field, if any. The printing procedure used for the base field (respectively, the mark field) depends on the value of CAESAR_BASE_AREA (resp. CAESAR_MARK_AREA) and is determined according to the rules specified for function CAESAR_USE_PRINT_FUNCTION_AREA_1() of the "area_1" library.

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

The above procedures CAESAR_OVERFLOW_SIGNAL_TABLE_1(), CAESAR_OVERFLOW_ABORT_TABLE_1(), and CAESAR_OVERFLOW_IGNORE_TABLE_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_TABLE_1.

The table is initially empty. The success and failure counters attached to the table are both initialized to 0.

.....

```
void CAESAR_DELETE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 *CAESAR_T;
   { ... }
```

This procedure frees the memory space corresponding to the table pointed to by *CAESAR_T using CAESAR_DELETE(). The items contained in the table and its associated hash-table are also freed. Afterwards, the NULL value is assigned to *CAESAR_T.

.....

```
void CAESAR_PURGE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure empties the table pointed to by CAESAR_T without deleting it. Each item contained in the table is freed using CAESAR_DELETE(). Afterwards, this table is exactly in the same state as

after its creation using `CAESAR_CREATE_TABLE_1()`.

. . . . . .

```
CAESAR_TYPE_INDEX_TABLE_1 CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns the number of items that have been put in the table pointed to by `CAESAR_T`, which is also equal to the index of the next item to be put in the table. This number is initialized to 0 and it is incremented every time the `CAESAR_PUT_TABLE_1()` function (see below) is called.

Note: This number is always less or equal to N, where N is the maximal number of items that can be stored in the table; the value of N depends on the actual value given to the formal parameter `CAESAR_LIMIT_SIZE` when the table was created using `CAESAR_CREATE_TABLE_1()`.

Note: When the table is empty, the result returned by this function is equal to zero. When the table is full, the result returned by this function is equal to N.

. . . . . .

```
CAESAR_TYPE_POINTER CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

If the table pointed to by `CAESAR_T` is not full, this function returns a pointer to the base field of the next item to be put in the table. This base field is initially undefined and must be assigned before calling some other functions of the "table_1" library (see below).

If the table pointed to by `CAESAR_T` is full, this function returns a pointer to the base field of a special item located beyond the table. It is permitted to modify (and subsequently consult) the contents of this base field, but invoking the `CAESAR_PUT_TABLE_1()` function (see below) will cause an overflow.

. . . . . .

```
CAESAR_TYPE_POINTER CAESAR_PUT_MARK_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

If there are no mark fields in the table (due to the initialization parameters supplied to `CAESAR_CREATE_TABLE_1()`), the result returned by `CAESAR_PUT_MARK_TABLE_1()` is always undefined.

If the table pointed to by `CAESAR_T` is not full, this function returns a pointer to the mark field of the next item to be put in the table. This mark field is always initialized to a bit string of 0's. It can be consulted and modified.

If the table pointed to by `CAESAR_T` is full, this function returns a pointer to the mark field of a special item located beyond the table. It is permitted to modify and consult the contents of this mark field, but invoking the `CAESAR_PUT_TABLE_1()` function (see below) will cause an overflow.

. . . . . .

```
void CAESAR_PUT_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure puts into the table pointed to by `CAESAR_T` the item whose base field is pointed to by `CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)` and whose mark field (if any) is pointed to by `CAESAR_PUT_MARK_TABLE_1 (CAESAR_T)`.

The base field must have been assigned before this procedure is called.

This procedure assumes that no other item in the table has the same base field. There is no attempt to check the validity of this assumption. It is therefore of the user's responsibility to ensure that this assumption is true. See also function `CAESAR_SEARCH_AND_PUT_TABLE_1()` below.

The hash-table associated to the table is updated to take into account the new item. To compute the hash-value for the base field, the hash-function associated with the table is used.

If the maximum number of items in the table was already reached when the procedure `CAESAR_PUT_TABLE_1()` was called, or in case of memory shortage, the overflow procedure associated with `CAESAR_T` is called with the actual parameter `CAESAR_T`.

Finally, `CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)` is incremented; `CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)` and `CAESAR_PUT_MARK_TABLE_1 (CAESAR_T)` are advanced, respectively, to the base field and the mark field (if any) of the next free item.

Note: the table is implemented in such a way that `CAESAR_PUT_BASE_TABLE_1()` and `CAESAR_PUT_MARK_TABLE_1()` always return a valid pointer, even if the table is already full. Overflow can only occur when `CAESAR_PUT_TABLE_1()` is called, but not when `CAESAR_PUT_BASE_TABLE_1()` or `CAESAR_PUT_MARK_TABLE_1()` are called.

. . . . . .

```
CAESAR_TYPE_BOOLEAN CAESAR_SEARCH_TABLE_1 (CAESAR_T, CAESAR_B, CAESAR_I, CAESAR_P)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_POINTER CAESAR_B;
   CAESAR_TYPE_INDEX_TABLE_1 *CAESAR_I;
   CAESAR_TYPE_POINTER *CAESAR_P;
   { ... }
```

This function determines if there exists, in the table pointed to by `CAESAR_T`, an item whose base field is equal to the byte string pointed to by `CAESAR_B`. Byte string comparisons are performed using the comparison function associated to the table. The search is done using the hash-function and hash-table associated to the table.

If so, this function returns `CAESAR_TRUE`. In this case, the index and the address of the item are respectively assigned to `*CAESAR_I` and `*CAESAR_P`. The success counter attached to the table is incremented.

If not, this function returns `CAESAR_FALSE`. In this case, both variables `*CAESAR_I` and `*CAESAR_P` are left unchanged. The failure counter attached to the table is incremented.

. . . . . .

```
CAESAR_TYPE_BOOLEAN CAESAR_SEARCH_AND_PUT_TABLE_1 (CAESAR_T, CAESAR_I, CAESAR_P)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_INDEX_TABLE_1 *CAESAR_I;
   CAESAR_TYPE_POINTER *CAESAR_P;
    { ... }
```

This function is a combination of the function `CAESAR_SEARCH_TABLE_1()` and the procedure `CAESAR_PUT_TABLE_1()` defined above.

The base field pointed to by `CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)` must have been assigned before this function is called.

It first determines if there exists, in the table pointed by `CAESAR_T`, an item whose base field is equal to the base field of the item pointed to by `CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)`. Byte string comparisons are performed using the comparison function associated to the table. The search is done using the hash-function and hash-table associated to the table.

If so, this function returns `CAESAR_TRUE`. In this case, the index and the address of the existing item are respectively assigned to `*CAESAR_I` and `*CAESAR_P`. The success counter attached to the table is incremented.

If not, this function returns `CAESAR_FALSE`. In this case, it puts into the table pointed to by `CAESAR_T` the item whose base field is pointed to by `CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)` and whose mark field (if any) is pointed to by `CAESAR_PUT_MARK_TABLE_1 (CAESAR_T)`. The hash-table associated to the table is updated to take into account the new item. The overflow procedure associated with the table is called if overflow occurs.

`CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)` is copied into `*CAESAR_I` and then incremented. `CAESAR_PUT_BASE_TABLE_1 (CAESAR_T)` is copied into `*CAESAR_P` and then advanced to the base field of the next free item. The failure counter attached to the table is incremented.

Note: formally the body of function `CAESAR_SEARCH_AND_PUT_TABLE_1()` could be defined as follows:

```
{
CAESAR_TYPE_BOOLEAN caesar_found;
caesar_found = CAESAR_SEARCH_TABLE_1 (CAESAR_T,
        CAESAR_PUT_BASE_TABLE_1 (CAESAR_T), CAESAR_I, CAESAR_P);
if (! caesar_found)
        {
        *CAESAR_I = CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T);
        *CAESAR_P = CAESAR_PUT_BASE_TABLE_1 (CAESAR_T);
        CAESAR_PUT_TABLE_1 (CAESAR_T);
        }
}
```

Practically, it is implemented differently, for efficiency reasons (the computation of the hash-value and the access through the hash-table are performed only once, not twice).

.....

```
CAESAR_TYPE_INDEX_TABLE_1 CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
```

```
   { ... }
```

This function returns the number of items that have been got from the table pointed to by `CAESAR_T`, which is also the index of the next item to be got from the table. This number is initialized to 0 and it is incremented every time the `CAESAR_GET_TABLE_1()` function (see below) is called.

Note: This number is always less or equal to the value returned by `CAESAR_PUT_INDEX_TABLE_1()`.

.....

```
CAESAR_TYPE_POINTER CAESAR_GET_BASE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a pointer to the base field of the next item to be got in the table pointed to by `CAESAR_T`.

This pointer can only be used if `CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)` is strictly less than `CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)`. In the opposite case, the result of this function is undefined (since it is not possible to get items that have not been put yet).

The base field pointed to by the result of `CAESAR_GET_BASE_TABLE_1()` can be consulted, but not modified.

.....

```
CAESAR_TYPE_POINTER CAESAR_GET_MARK_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a pointer to the mark field of the next item to be got in the table pointed to by `CAESAR_T`. If there are no mark fields in the table (due to the initialization parameters supplied to `CAESAR_CREATE_TABLE_1()`) the result is undefined.

This pointer can only be used if `CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)` is strictly less than `CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)`. In the opposite case, the result of this function is undefined (since it is not possible to get items that have not been put yet).

The mark field pointed to by the result of `CAESAR_GET_MARK_TABLE_1()` can be either consulted or modified.

.....

```
void CAESAR_GET_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure increments `CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)`; it advances, respectively, `CAESAR_GET_BASE_TABLE_1 (CAESAR_T)` and `CAESAR_GET_MARK_TABLE_1 (CAESAR_T)` to the base field and the mark field (if any) of the next free item.

If `CAESAR_GET_INDEX_TABLE_1 (CAESAR_T)` is equal to `CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)` when the procedure `CAESAR_GET_TABLE_1()` is called, the result is undefined.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_EMPTY_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a value different from 0 if the table pointed to by `CAESAR_T` is empty, and 0 otherwise. `CAESAR_EMPTY_TABLE_1 (CAESAR_T)` is always equivalent to:

```
CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T) == 0
```

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_FULL_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a value different from 0 if the table pointed to by `CAESAR_T` is full, and 0 otherwise. `CAESAR_FULL_TABLE_1 (CAESAR_T)` is always equivalent to:

`CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T) ==` N

where N denotes the maximum number of items that the table can contain; the value of N depends on the actual value given to the formal parameter `CAESAR_LIMIT_SIZE` when the table was created using `CAESAR_CREATE_TABLE_1()`.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_EXPLORED_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns a value different from 0 if the get index and the put index are identical, or 0 otherwise. `CAESAR_EXPLORED_TABLE_1 (CAESAR_T)` is always equivalent to:

```
CAESAR_GET_INDEX_TABLE_1 (CAESAR_T) == CAESAR_PUT_INDEX_TABLE_1 (CAESAR_T)
```

.....

```
void CAESAR_RETRIEVE_I_B_TABLE_1 (CAESAR_T, CAESAR_I, CAESAR_B)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_INDEX_TABLE_1 CAESAR_I;
```

```
   CAESAR_TYPE_POINTER *CAESAR_B;
   { ... }
```

This procedure computes the address of the base field of the item with index `CAESAR_I` in the table pointed to by `CAESAR_T`. This address is assigned to `*CAESAR_B`.

If `CAESAR_I` is greater or equal to the value returned by `CAESAR_PUT_INDEX_TABLE_1()`, a `NULL` pointer is assigned to `*CAESAR_B`.

.....

```
void CAESAR_RETRIEVE_I_M_TABLE_1 (CAESAR_T, CAESAR_I, CAESAR_M)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_INDEX_TABLE_1 CAESAR_I;
   CAESAR_TYPE_POINTER *CAESAR_M;
   { ... }
```

This procedure computes the address of the mark field of the item with index `CAESAR_I` in the table pointed to by `CAESAR_T`. This address is assigned to `*CAESAR_M`.

If `CAESAR_I` is greater or equal to the value returned by `CAESAR_PUT_INDEX_TABLE_1()`, a `NULL` pointer is assigned to `*CAESAR_M`.

If there are no mark fields in the table (due to the initialization parameters supplied to `CAESAR_CREATE_TABLE_1()`) the effect of this procedure is undefined.

.....

```
void CAESAR_RETRIEVE_I_BM_TABLE_1 (CAESAR_T, CAESAR_I, CAESAR_B, CAESAR_M)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_INDEX_TABLE_1 CAESAR_I;
   CAESAR_TYPE_POINTER *CAESAR_B;
   CAESAR_TYPE_POINTER *CAESAR_M;
   { ... }
```

This procedure computes the address of the base field of the item with index `CAESAR_I` in the table pointed to by `CAESAR_T`. This address is assigned to `*CAESAR_B`.

It also computes the address of the mark field of the item with index `CAESAR_I` in the table pointed to by `CAESAR_T`. This address is assigned to `*CAESAR_M`.

If `CAESAR_I` is greater or equal to the value returned by `CAESAR_PUT_INDEX_TABLE_1()`, a `NULL` pointer is assigned to `*CAESAR_B` and `*CAESAR_M`.

If there are no mark fields in the table (due to the initialization parameters supplied to `CAESAR_CREATE_TABLE_1()`) the effect of this procedure is undefined.

.....

```
void CAESAR_RETRIEVE_B_I_TABLE_1 (CAESAR_T, CAESAR_B, CAESAR_I)
CAESAR_TYPE_TABLE_1 CAESAR_T;
CAESAR_TYPE_POINTER CAESAR_B;
```

```
CAESAR_TYPE_INDEX_TABLE_1 *CAESAR_I;
   { ... }
```

This procedure computes the index of the item whose base field, in the table pointed to by CAESAR_T, is pointed to by CAESAR_B. This index is assigned to *CAESAR_I.

If no item stored in the table has a base field at address CAESAR_B, the CAESAR_NULL_INDEX_TABLE_1 value is assigned to *CAESAR_I.

.....

```
void CAESAR_RETRIEVE_M_I_TABLE_1 (CAESAR_T, CAESAR_M, CAESAR_I)
CAESAR_TYPE_TABLE_1 CAESAR_T;
CAESAR_TYPE_POINTER CAESAR_M;
CAESAR_TYPE_INDEX_TABLE_1 *CAESAR_I;
   { ... }
```

This procedure computes the index of the item whose mark field, in the table pointed to by CAESAR_T, is pointed to by CAESAR_M. This index is assigned to *CAESAR_I.

If no item stored in the table has a mark field at address CAESAR_M, the CAESAR_NULL_INDEX_TABLE_1 value is assigned to *CAESAR_I.

If there are no mark fields in the table (due to the initialization parameters supplied to CAESAR_CREATE_TABLE_1()) the effect of this procedure is undefined.

.....

```
void CAESAR_RETRIEVE_B_M_TABLE_1 (CAESAR_T, CAESAR_B, CAESAR_M)
CAESAR_TYPE_TABLE_1 CAESAR_T;
CAESAR_TYPE_POINTER CAESAR_B;
CAESAR_TYPE_POINTER *CAESAR_M;
   { ... }
```

This procedure computes, for the table pointed to by CAESAR_T, the address of the mark field of the item whose base field is pointed to by CAESAR_B. This address is assigned to *CAESAR_M.

If no item stored in the table has a base field at address CAESAR_B, the effect of this procedure is undefined.

If there are no mark fields in the table (due to the initialization parameters supplied to CAESAR_CREATE_TABLE_1()) the effect of this procedure is undefined.

.....

```
void CAESAR_RETRIEVE_M_B_TABLE_1 (CAESAR_T, CAESAR_M, CAESAR_B)
CAESAR_TYPE_TABLE_1 CAESAR_T;
CAESAR_TYPE_POINTER CAESAR_M;
CAESAR_TYPE_POINTER *CAESAR_B;
   { ... }
```

This procedure computes, for the table pointed to by CAESAR_T, the address of the base field of the

item whose mark field is pointed to by `CAESAR_M`. This address is assigned to `*CAESAR_B`.

If no item stored in the table has a mark field at address `CAESAR_M`, the effect of this procedure is undefined.

If there are no mark fields in the table (due to the initialization parameters supplied to `CAESAR_CREATE_TABLE_1()`) the effect of this procedure is undefined.

.....

```
CAESAR_TYPE_NATURAL CAESAR_FAILURE_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns the value of the failure counter of the table pointed to by `CAESAR_T`, i.e., the number of searches that failed.

.....

```
CAESAR_TYPE_NATURAL CAESAR_SUCCESS_TABLE_1 (CAESAR_T)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This function returns the value of the success counter of the table pointed to by `CAESAR_T`, i.e., the number of searches that succeeded.

.....

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_TABLE_1 (CAESAR_T, CAESAR_FORMAT)
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the table pointed to by `CAESAR_T` will be printed by the procedure `CAESAR_PRINT_TABLE_1()` (see below).

Currently, the following formats are available:

- With format 0, statistical information about the table is displayed such as: the number of items put, the number of items got, the size in bytes, the success counter, the failure counter, etc.
- With format 1, all items in the table are printed, sorted by increasing indexes. For each item, the corresponding index is displayed; the base field and the mark field (if any) are also displayed using the printing procedure associated to the table.
- With format 2, all items in the table are printed, sorted by increasing indexes. For each item, the index, the address, and the corresponding hash-value are displayed; the base field and the mark field (if any) are also displayed using the printing procedure associated to the table. Informations concerning the associated hash-table are also displayed. This format is mainly intended for debugging purpose.
- (no other format available yet)

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

When called with `CAESAR_FORMAT` between 0 and 2, this fonction sets the current format of `CAESAR_T` 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_T` 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_T`. If `CAESAR_FORMAT` is equal to the constant `CAESAR_MAXIMAL_FORMAT`, the result is the maximal format value (i.e., 2). In all other cases, the effect of this function is undefined.

. . . . . .

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_TABLE_1 ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as it might be removed from future versions of the Open/Cæsar. Use function `CAESAR_FORMAT_TABLE_1()` instead, called with argument `CAESAR_MAXIMAL_FORMAT`.

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

. . . . . .

```
void CAESAR_PRINT_TABLE_1 (CAESAR_FILE, CAESAR_T)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_TABLE_1 CAESAR_T;
   { ... }
```

This procedure prints to file `CAESAR_FILE` a text containing information about the table pointed to by `CAESAR_T`. The nature of the information is determined by the current format of the table pointed to by `CAESAR_T`.

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

. . . . . .

Chapter 12

The "cache_1" library (version 1.2)

by Radu Mateescu

12.1 Purpose

The "cache_1" library provides primitives for storing arbitrary data items in caches, which can be organized hierarchically. It can be used to speed up the execution of applications that involve costly search operations (by storing data items in caches for fast retrievals), and also to reduce the memory consumption of state space exploration algorithms (by storing a certain amount of states in caches).

12.2 Usage

The "cache_1" library consists of:

- a predefined header file "`caesar_cache_1.h`";
- the precompiled library file "`libcaesar.a`", which implements the features described in "`caesar_cache_1.h`".

Note: The "cache_1" library is a software layer built above the primitives offered by the "standard", "area_1", and "hash" libraries, and by the OPEN/CÆSAR graph module. It follows as close as possible the principles and features of the "table_1" library.

12.3 Description

A "cache" is basically a set containing a fixed number of items.

Each item in the cache is basically a byte string of fixed size. All items in a given cache have the same size. An item can be considered as a tuple with two fields, whose size and contents are freely determined by the user:

- (1) a "base" field, that is a byte string of fixed size. In a given cache, all base fields have the same size. This size must be greater than zero.

Sometimes, the base field contains a state (as defined in the graph module). However, this is not mandatory, and base fields can contain other information than states.

- (2) a "mark" field, that is a byte string whose size and contents are freely determined by the user. In a given cache, all mark fields have the same size, which must be greater or equal to zero. Pointers to mark fields will be considered as values of type `CAESAR_TYPE_POINTER`; mark fields are always aligned on appropriate boundaries so that the user can put any information in these fields without alignment problem.

The user also determines the nature of the data stored in these fields, which is not meaningful to the "cache_1" library.

A cache is organized as a collection of "subcaches", each one containing a fixed number of items. A cache can contain no more than a maximum of P subcaches, where currently $P = 256$. In a cache containing N subcaches, each subcache is assigned an unique index in the range $0..N-1$. The subcaches are linked hierarchically by a parent relation, which links a subcache to its "children" subcaches. The parent relation is an acyclic relation having as root element the subcache of index 0, denoted "root subcache" in the sequel. All the other subcaches are assumed to be descendants of the root subcache, i.e., they should be reachable from the root subcache via the parent relation. A subcache without children subcaches is called a "leaf" subcache. The parent relation linking the subcaches of a cache is not necessarily static, but can dynamically change during the usage of the cache.

A cache can contain no more than a maximum of M items. Currently, $M = 2^{29} = 536,870,912$ on 32-bit machines and $M = 2^{34} = 17,179,869,184$ on 64-bit machines. But, for each cache, the user can also limit the maximal number of items to a lesser bound $K < M$.

To each cache is associated its current global date, which is the number of modifying operations performed on the cache since it was created. The operations modifying the status of a cache are the following: putting an item into the cache, "hitting" an item (i.e., searching and finding the item) in the cache, deleting an item from the cache, and updating an item present in the cache. In the sequel, whenever this is understood from the context, we will use the term "date" to designate the current global date of a cache. To each subcache of a cache is associated its current local date, i.e., the date when the last modifying operation was performed on an item present in the subcache.

To each item present in a cache are associated the following fields:

- the date when the item was put into the cache;
- the date of the last access to the item by a put, a hit, or an update operation;
- the number of hits at the item since it was put into the cache.

All these fields can be accessed from the address of the item (i.e., a pointer to the memory location where the item is stored in the cache).

Invariant property 1: all items present in a cache have different dates when they were put into the cache. Therefore, the date when an item was put into the cache can serve as unique identification number for the item.

Invariant property 2: all items present in a cache have different last access dates (but several items may have the same number of hits).

Invariant property 3: it is not allowed to modify the base field of any item in the cache. But it is possible to modify the mark field of any item.

Each item of a cache can be accessed by using its address or its base field. The cache data structure establishes a correspondence between these two data. Indeed:

- given an address, one can retrieve the base field and the mark field of the corresponding item;
- given a base field, one can retrieve the address and the mark field of the corresponding item.

Retrieving the address of an item from its base field involves some associative search. To allow fast retrievals, a hash-table is associated to each cache. This is quite transparent from the user's point of view. Only the base field is taken into account when computing the hash-value and comparing items; the mark field is not meaningful for the search.

To each cache are associated two counters recording the number of search and hit operations performed on the cache, respectively. To each subcache of a cache is associated a counter recording the number of hit operations at (items of) that subcache.

The operation of putting an item E into a cache proceeds as follows. The item is always put into the root subcache (of index 0). When this subcache becomes full, the item E replaces the smallest item E1 contained in this subcache according to the order relation underlying the replacement strategy associated to this subcache. The item E1 is put into one of the child subcaches (of index I1) of the root subcache, determined by the parent relation between subcaches and by the contents of the item E1. If the subcache of index I1 is also full, then the item E1 replaces the smallest item E2 contained in this subcache, and the item E2 will be in turn put into a child subcache (of index I2) of the subcache of index I1, and so on. This process continues along a sequence of subcaches in the hierarchy until either it reaches a subcache of index Ij that is not full (and can therefore accept the current item Ej to be put into it), or it reaches a leaf subcache of index Ik that is full. In the latter case, the item Ek+1 replaced in the leaf subcache is stored temporarily in a field associated to the cache until the next put operation on the cache causes another item El+1 to be replaced in some leaf subcache of index Il that is already full; the item Ek+1 is then deleted and replaced by El+1.

The parent relation between the subcaches of a cache is assumed to be acyclic (in order to ensure the termination of put operations), but this condition cannot be checked at the creation of the cache because the parent relation may change dynamically during execution. Instead, this condition is checked at each put operation, and if a cycle between subcaches is detected, then the operation is stopped and an appropriate error code is set.

The "cache_1" library supports applications involving dynamic creation of caches. When a put operation performed on a cache C1 causes a replacement to take place, another cache C2 can be created and the last item replaced in C1 (which can be inspected using the `CAESAR_LAST_ITEM_REPLACED_CACHE_1()` procedure below) can be put into C2. This enables to build hierarchies of caches similar to the hierarchies of subcaches present in individual caches. It is the user's responsibility to ensure that a put operation performed on a cache of a hierarchy does not cause a cycle of put operations on the other caches of the hierarchy.

12.4 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_CACHE_1;
```

This type denotes a pointer to the concrete representation of a cache. This representation is supposed to be "opaque".

. . . . . .

```
typedef CAESAR_TYPE_NATURAL
   (*CAESAR_TYPE_NATURAL_FUNCTION_CACHE_1) (CAESAR_TYPE_NATURAL);
```

This type denotes a pointer to a function which takes as parameter a natural number (index of a subcache) and returns a natural number (size or percentage of the subcache).

.....

```
typedef CAESAR_TYPE_INTEGER
   (*CAESAR_TYPE_ORDER_FUNCTION_CACHE_1) (CAESAR_TYPE_CACHE_1,
      CAESAR_TYPE_POINTER, CAESAR_TYPE_POINTER);
```

This type denotes a pointer to a function which takes as parameters a cache and two pointers to items (supposed to be contained in the cache), and returns an integer number indicating whether the first item is smaller than, equal to, or greater than the second one modulo an order relation.

.....

```
typedef CAESAR_TYPE_NATURAL
   (*CAESAR_TYPE_SUBCACHE_FUNCTION_CACHE_1) (CAESAR_TYPE_CACHE_1,
      CAESAR_TYPE_NATURAL, CAESAR_TYPE_POINTER);
```

This type denotes a pointer to a function which takes as parameters a cache, a natural number (index of a subcache) and a pointer to an item (the last item replaced in the subcache), and returns a natural number (index of the child subcache in which the replaced item will be put).

.....

```
typedef void
   (*CAESAR_TYPE_CLEANUP_FUNCTION_CACHE_1) (CAESAR_TYPE_POINTER);
```

This type denotes a pointer to a procedure which takes as parameter a pointer to an item and cleans up the contents of the item (see the procedure `CAESAR_CREATE_CACHE_1()` below).

.....

```
typedef enum {
     CAESAR_NONE_CACHE_1,
     CAESAR_CYCLIC_CACHE_1
}    CAESAR_TYPE_ERROR_CACHE_1;
```

This enumerated type defines the error codes produced as a side effect by calls to the procedure `CAESAR_PUT_CACHE_1()` or to the function `CAESAR_SEARCH_AND_PUT_CACHE_1()` (see below), which put an item into a cache. The error codes have the following meaning:

- `CAESAR_NONE_CACHE_1` indicates that the put operation was performed successfully.

- `CAESAR_CYCLIC_CACHE_1` indicates the existence of a cycle in the parent relation between subcaches, which would cause the put operation to loop indefinitely when the subcaches present on that cycle are full.

Note: The error code produced by a call to the procedure `CAESAR_PUT_CACHE_1()` or to the function `CAESAR_SEARCH_AND_PUT_CACHE_1()` can be obtained by using the function `CAESAR_STATUS_PUT_CACHE_1()` (see below).

.....

```
CAESAR_TYPE_INTEGER CAESAR_LRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "least recently used" (LRU) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the date of the last access to the first item is smaller than, equal to, or greater than the date of the last access to the second item, respectively.

.....

```
CAESAR_TYPE_INTEGER CAESAR_MRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "most recently used" (MRU) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the date of the last access to the first item is greater than, equal to, or smaller than the date of the last access to the second item, respectively.

.....

```
CAESAR_TYPE_INTEGER CAESAR_LRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "least recently put" (LRP) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the date when the first item was put into the cache is smaller than, equal to, or greater than the date when the second item was put into the cache, respectively.

.....

```
CAESAR_TYPE_INTEGER CAESAR_MRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "most recently put" (MRP) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the date when the first item was put into the cache is greater than, equal to, or smaller than the date when the second item was put into the cache, respectively.

.....

```
CAESAR_TYPE_INTEGER CAESAR_LFU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "least frequently used" (LFU) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the number of hits at the first item is smaller than, equal to, or greater than the number of hits at the second item, respectively.

.....

```
CAESAR_TYPE_INTEGER CAESAR_MFU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
```

```
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "most frequently used" (MFU) replacement strategy.

According to this order relation, an item is smaller than, equal to, or greater than another item if the number of hits at the first item is greater than, equal to, or smaller than the number of hits at the second item, respectively.

. . . . . .

```
CAESAR_TYPE_INTEGER CAESAR_RND_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "random" (RND) replacement strategy.

This order relation is implemented by computing for every item a random cost, i.e., a natural number randomly generated by taking into account the seed associated to the cache pointed to by `CAESAR_C` (see the procedure `CAESAR_SEED_RND_CACHE_1()` below), the current global date of the cache, and the base field of the item. According to this relation, an item is smaller than, equal to, or greater than another item if the random cost of the first item is smaller than, equal to, or greater than the random cost of the second item, respectively.

. . . . . .

```
CAESAR_TYPE_INTEGER CAESAR_LFU_LRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "least frequently used, then least recently used" (LFU_LRU) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is smaller than the date of the last access to the second one;
- an item is equal to another one if the number of hits and the date of the last access are the same for both items;

- an item is greater than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is greater than the date of the last access to the second one.

In other words, the LFU_LRU replacement strategy consists in applying first the LFU, then the LRU replacement strategies.

.....

```
CAESAR_TYPE_INTEGER CAESAR_LFU_MRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "least frequently used, then most recently used" (LFU_MRU) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is greater than the date of the last access to the second one;
- an item is equal to another one if the number of hits and the date of the last access are the same for both items;
- an item is greater than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is smaller than the date of the last access to the second one.

In other words, the LFU_MRU replacement strategy consists in applying first the LFU, then the MRU replacement strategies.

.....

```
CAESAR_TYPE_INTEGER CAESAR_LFU_LRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "least frequently used, then least recently put" (LFU_LRP) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is smaller than the date when the second one was put into the cache;
- an item is equal to another one if the number of hits and the date when they were put into the cache are the same for both items;
- an item is greater than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is greater than the date when the second one was put into the cache.

In other words, the LFU_LRP replacement strategy consists in applying first the LFU, then the LRP replacement strategies.

.....

```
CAESAR_TYPE_INTEGER CAESAR_LFU_MRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "least frequently used, then most recently put" (LFU_MRP) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is greater than the date when the second one was put into the cache;
- an item is equal to another one if the number of hits and the date when they were put into the cache are the same for both items;
- an item is greater than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is smaller than the date when the second one was put into the cache.

In other words, the LFU_MRP replacement strategy consists in applying first the LFU, then the MRP replacement strategies.

.....

```
CAESAR_TYPE_INTEGER CAESAR_MFU_LRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
```

```
CAESAR_TYPE_CACHE_1 CAESAR_C;
CAESAR_TYPE_POINTER CAESAR_B1;
CAESAR_TYPE_POINTER CAESAR_B2;
{ ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "most frequently used, then least recently used" (MFU_LRU) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is smaller than the date of the last access to the second one;
- an item is equal to another one if the number of hits and the date of the last access are the same for both items;
- an item is greater than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is greater than the date of the last access to the second one.

In other words, the MFU_LRU replacement strategy consists in applying first the MFU, then the LRU replacement strategies.

.....

```
CAESAR_TYPE_INTEGER CAESAR_MFU_MRU_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "most frequently used, then most recently used" (MFU_MRU) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is greater than the date of the last access to the second one;
- an item is equal to another one if the number of hits and the date of the last access are the same for both items;
- an item is greater than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date of the last access to the first item is smaller than the date of the last access to the second one.

In other words, the MFU_MRU replacement strategy consists in applying first the MFU, then the MRU replacement strategies.

. . . . . .

```
CAESAR_TYPE_INTEGER CAESAR_MFU_LRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "most frequently used, then least recently put" (MFU_LRP) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is smaller than the date when the second one was put into the cache;
- an item is equal to another one if the number of hits and the date when they were put into the cache are the same for both items;
- an item is greater than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is greater than the date when the second one was put into the cache.

In other words, the MFU_LRP replacement strategy consists in applying first the MFU, then the LRP replacement strategies.

. . . . . .

```
CAESAR_TYPE_INTEGER CAESAR_MFU_MRP_ORDER_CACHE_1 (CAESAR_C, CAESAR_B1, CAESAR_B2)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B1;
   CAESAR_TYPE_POINTER CAESAR_B2;
   { ... }
```

This function returns an integer number which is smaller than, equal to, or greater than 0 depending whether the item pointed to by `CAESAR_B1` is smaller than, equal to, or greater than the item pointed to by `CAESAR_B2` according to the order relation underlying the "most frequently used, then most recently put" (MFU_MRP) replacement strategy.

According to this order relation:

- an item is smaller than another item if the number of hits at the first item is greater than the number of hits at the second one, or the number of hits is the same for both items but the date

when the first item was put into the cache is greater than the date when the second one was put into the cache;

- an item is equal to another one if the number of hits and the date when they were put into the cache are the same for both items;
- an item is greater than another item if the number of hits at the first item is smaller than the number of hits at the second one, or the number of hits is the same for both items but the date when the first item was put into the cache is smaller than the date when the second one was put into the cache.

In other words, the MFU_MRP replacement strategy consists in applying first the MFU, then the MRP replacement strategies.

.....

```
void CAESAR_SEED_RND_CACHE_1 (CAESAR_C, CAESAR_SEED)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_SEED;
   { ... }
```

This procedure initializes the seed for the random number generator associated to the cache pointed to by `CAESAR_C` with the value of `CAESAR_SEED`. The seed is used for computing the random costs associated to the items contained in the subcaches of the cache that are equipped with the RND replacement strategy.

Note: The value of the seed associated to a cache is set by default to 0 when the cache is created by invoking the procedure `CAESAR_CREATE_CACHE_1()` (see below).

.....

```
void CAESAR_CREATE_CACHE_1 (CAESAR_C,
                            CAESAR_NUMBER_OF_SUBCACHES,
                            CAESAR_LIMIT_SIZE,
                            CAESAR_SUBCACHE_SIZE,
                            CAESAR_SUBCACHE_PERCENTAGE,
                            CAESAR_SUBCACHE_ORDER,
                            CAESAR_SUBCACHE_CHILD,
                            CAESAR_BASE_AREA,
                            CAESAR_MARK_AREA,
                            CAESAR_HASH_SIZE,
                            CAESAR_PRIME,
                            CAESAR_COMPARE,
                            CAESAR_HASH,
                            CAESAR_PRINT,
                            CAESAR_CLEANUP,
                            CAESAR_INFO)
   CAESAR_TYPE_CACHE_1 *CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_SUBCACHES;
   CAESAR_TYPE_NATURAL CAESAR_LIMIT_SIZE;
```

```
CAESAR_TYPE_NATURAL_FUNCTION_CACHE_1 CAESAR_SUBCACHE_SIZE;
CAESAR_TYPE_NATURAL_FUNCTION_CACHE_1 CAESAR_SUBCACHE_PERCENTAGE;
CAESAR_TYPE_ORDER_FUNCTION_CACHE_1 (*CAESAR_SUBCACHE_ORDER)
   (CAESAR_TYPE_NATURAL);
CAESAR_TYPE_SUBCACHE_FUNCTION_CACHE_1 (*CAESAR_SUBCACHE_CHILD)
   (CAESAR_TYPE_NATURAL);
CAESAR_TYPE_AREA_1 CAESAR_BASE_AREA;
CAESAR_TYPE_AREA_1 CAESAR_MARK_AREA;
CAESAR_TYPE_NATURAL CAESAR_HASH_SIZE;
CAESAR_PROMOTE_TO_INT (CAESAR_TYPE_BOOLEAN) CAESAR_PRIME;
CAESAR_TYPE_COMPARE_FUNCTION CAESAR_COMPARE;
CAESAR_TYPE_HASH_FUNCTION CAESAR_HASH;
CAESAR_TYPE_PRINT_FUNCTION CAESAR_PRINT;
CAESAR_TYPE_CLEANUP_FUNCTION_CACHE_1 CAESAR_CLEANUP;
CAESAR_TYPE_POINTER CAESAR_INFO;
{ ... }
```

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

Note: Because `CAESAR_TYPE_CACHE_1` is a pointer type, any variable `CAESAR_C` of type `CAESAR_TYPE_CACHE_1` must be allocated before used, for instance using:

```
CAESAR_CREATE_CACHE_1 (&CAESAR_C, ...);
```

The value of `CAESAR_NUMBER_OF_SUBCACHES` determines the number of subcaches contained in the cache. Each subcache will be assigned an unique index in the range 0..`CAESAR_NUMBER_OF_SUBCACHES`-1. If the value of `CAESAR_NUMBER_OF_SUBCACHES` is zero or greater than P, the effect is undefined.

The value of `CAESAR_LIMIT_SIZE` determines the maximal number of items that can be stored in the cache. It must be less or equal to M. If it is equal to zero, it is replaced by the default value M.

Note: in order to spare memory, the value of `CAESAR_LIMIT_SIZE` (which is an upper bound on the number of items to be inserted in the cache) should be as small as possible. This can only be done if the user has some knowledge about the way the cache will be used.

The actual value of the formal parameter `CAESAR_SUBCACHE_SIZE` will be stored and associated to the cache pointed to by `*CAESAR_C`. It will be used to assign to each subcache its corresponding size, i.e., the maximal number of items the subcache can contain.

Precisely, the actual value of `CAESAR_SUBCACHE_SIZE` should be a pointer to a function with a parameter `caesar_index` that returns the size of the subcache of index `caesar_index`, where `caesar_index` is in the range 0..`CAESAR_NUMBER_OF_SUBCACHES` $-$ 1. If the value returned by `CAESAR_SUBCACHE_SIZE` for some index `caesar_index` is greater than zero, then the size of the subcache of index `caesar_index` is set to this value. If the value returned is zero, the size of the subcache of index `caesar_index` is unspecified and will be determined by the percentage returned by the `CAESAR_SUBCACHE_PERCENTAGE` parameter (see below). The sum of the sizes returned by `CAESAR_SUBCACHE_SIZE` for all subcaches, noted L, must be less or equal to `CAESAR_LIMIT_SIZE`.

The actual value of the formal parameter `CAESAR_SUBCACHE_PERCENTAGE` will be stored and associated to the cache pointed to by `*CAESAR_C`. It will be used to assign to each subcache its corresponding percentage, i.e., a natural number in the range 0..100 determining the size of the subcache with respect to the size of the other subcaches.

Precisely, the actual value of CAESAR_SUBCACHE_PERCENTAGE should be a pointer to a function with a parameter caesar_index that returns the percentage of the subcache of index caesar_index, where caesar_index is in the range 0..CAESAR_NUMBER_OF_SUBCACHES − 1. Several configurations are possible, enabling the setup of subcache sizes in a flexible manner:

- If the value returned by CAESAR_SUBCACHE_SIZE is greater than zero for all subcaches, then the values returned by CAESAR_SUBCACHE_PERCENTAGE are ignored because all subcache sizes are determined by CAESAR_SUBCACHE_SIZE.
- If there is some subcache with unspecified size (i.e., for which CAESAR_SUBCACHE_SIZE returns zero), then the size of the cache pointed to by *CAESAR_C is set to CAESAR_LIMIT_SIZE. Let R be the number of subcaches with unspecified size. The sizes of these subcaches are determined based on the percentages returned by CAESAR_SUBCACHE_PERCENTAGE. Two situations are possible:

  (1) If the value returned by CAESAR_SUBCACHE_PERCENTAGE is zero for all subcaches with unspecified size, then the size of each of these subcaches is set to $(\texttt{CAESAR\_LIMIT\_SIZE} - L)/R$.

  (2) If there is some cache with unspecified size for which the value returned by CAESAR_SUBCACHE_PERCENTAGE is greater than zero, then the size of a subcache with percentage F is set to $(\texttt{CAESAR\_LIMIT\_SIZE} - L) * F/100$. All the percentages returned by CAESAR_SUBCACHE_PERCENTAGE for the subcaches with unspecified size must be greater than zero, and the sum of all these percentages must be equal to 100.

The actual value of the formal parameter CAESAR_SUBCACHE_ORDER will be stored and associated to the cache pointed to by *CAESAR_C. It will be used to assign to each subcache a function implementing the order relation underlying the replacement strategy associated to the subcache.

Precisely, the actual value of CAESAR_SUBCACHE_ORDER should be a pointer to a function with a parameter caesar_index that returns a function implementing an order relation, where caesar_index is in the range 0..CAESAR_NUMBER_OF_SUBCACHES − 1. The value returned by CAESAR_SUBCACHE_ORDER should be a pointer to a function with three parameters caesar_cache, caesar_base_1, caesar_base_2. This function returns a value smaller than, equal to, or greater than 0 if the base field pointed to by caesar_base_1 is smaller than, equal to, or greater than the base field pointed to by caesar_base_2. The items whose base fields are pointed to by caesar_base_1 and caesar_base_2 are supposed to be contained in the cache pointed to by caesar_cache, which is always set to the value of *CAESAR_C, i.e., a pointer to the cache currently created. Examples of functions that can be returned by CAESAR_SUBCACHE_ORDER are those implementing the order relations underlying the predefined replacement strategies, namely CAESAR_LRU_ORDER_CACHE_1(), CAESAR_MRU_ORDER_CACHE_1(), etc.

The actual value of the formal parameter CAESAR_SUBCACHE_CHILD will be stored and associated to the cache pointed to by *CAESAR_C. It will be used to assign to each subcache a function implementing the parent relation that defines the child subcaches of that subcache.

Precisely, the actual value of CAESAR_SUBCACHE_CHILD should be a pointer to a function with a parameter caesar_index that returns a function implementing a parent relation, where caesar_index is in the range 0..CAESAR_NUMBER_OF_SUBCACHES−1. The value returned by CAESAR_SUBCACHE_CHILD should be a pointer to a function with three parameters caesar_cache, caesar_index, caesar_base. The parameter caesar_cache is always set to the value of *CAESAR_C, i.e., a pointer to the cache currently created. The parameter caesar_base is a pointer to the last item replaced in the subcache of index caesar_index when the current put operation performed on the cache pointed to by caesar_cache entailed a put operation on the subcache of index caesar_index, which was already full. This function returns, for the subcache of index caesar_index, the index of its child subcache

in which the item pointed to by caesar_base will be put. If the index returned by this function is greater or equal to CAESAR_NUMBER_OF_SUBCACHES, then the subcache of index caesar_index is considered to be a leaf subcache, and the last item replaced pointed to by caesar_base will be temporarily stored in a field of the cache until the next replacement takes place in the cache (see the CAESAR_LAST_ITEM_REPLACED_CACHE_1() procedure below).

For example, the following function implements a parent relation corresponding to a stream of subcaches, i.e., a hierarchy in which each subcache of index I has a single child subcache of index I+1:

```
CAESAR_TYPE_NATURAL caesar_child (caesar_cache, caesar_index, caesar_base)
CAESAR_TYPE_CACHE_1 caesar_cache;
CAESAR_TYPE_NATURAL caesar_index;
CAESAR_TYPE_POINTER caesar_base;
{
   return (caesar_index + 1);
}
```

This function does not use the caesar_cache and the caesar_base parameters. However, general user-defined functions can implement parent relations that may change dynamically depending on the current status of the cache pointed to by caesar_cache, the current status of its subcache of index caesar_index and/or the contents of the item pointed to by caesar_base.

The value of CAESAR_BASE_AREA determines the (constant) size and (constant) alignment factor of the base field in the cache. In the particular case where base fields are used to store states (resp. labels, strings), one must give the value CAESAR_STATE_AREA_1() (resp. CAESAR_LABEL_AREA_1(), CAESAR_STRING_AREA_1()) to the formal parameter CAESAR_BASE_AREA.

The value of CAESAR_MARK_AREA determines the (constant) size and (constant) alignment factor of the mark field according to the specifications of the "area˙1" library. In particular, if CAESAR_MARK_AREA is equal to CAESAR_EMPTY_AREA_1(), there will be no mark field in the cache.

Each item in the cache will be represented as a byte string of fixed size caesar_item_size, such that caesar_item_size is greater or equal to caesar_base_size + caesar_mark_size, where caesar_base_size denotes the size (in bytes) of the base field (i.e., CAESAR_SIZE_AREA_1 (CAESAR_BASE_AREA)) and where caesar_mark_size denotes the size (in bytes) of the mark field, if any (i.e., CAESAR_SIZE_AREA_1 (CAESAR_MARK_AREA)).

An item in the cache contains not only the base field and the mark field, but also "padding" bytes that may be inserted between the base and mark fields to ensure that these fields are correctly aligned according to CAESAR_ALIGNMENT_AREA_1 (CAESAR_BASE_AREA) and CAESAR_ALIGNMENT_AREA_1 (CAESAR_MARK_AREA).

The value of CAESAR_HASH_SIZE determines the number of entries in the hash-table associated to the cache. If it is equal to zero, it is replaced with a default value greater than zero.

If the value of CAESAR_PRIME is equal to CAESAR_TRUE and if the value of CAESAR_HASH_SIZE is not a prime number, this value will be replaced by the nearest smaller prime number (since some hash functions require prime modulus). Otherwise, the value of CAESAR_HASH_SIZE will be kept unchanged.

The actual value of the formal parameter CAESAR_COMPARE will be stored and associated to the cache pointed to by *CAESAR_C. It will be used as a comparison function when it is necessary to decide whether two base fields are equal or not.

Precisely, the actual value of CAESAR_COMPARE should be a pointer to a comparison function with two parameters caesar_base_1 and caesar_base_2 that returns CAESAR_TRUE if the two base fields pointed to by caesar_base_1 and caesar_base_2 are equal.

If the actual value of the formal parameter `CAESAR_COMPARE` is `NULL`, it is replaced by a pointer to a default comparison function that depends on the value of `CAESAR_BASE_AREA` and is determined according to the rules specified for function `CAESAR_USE_COMPARE_FUNCTION_AREA_1()` of the "area˙1" library.

The actual value of the formal parameter `CAESAR_HASH` will be stored and associated to the cache pointed to by `*CAESAR_C`. It will be used as a hash-function when it is necessary to compute a hash-value for searching or inserting an item in the cache.

Precisely, the actual value of `CAESAR_HASH` should be a pointer to a hash function with two parameters `caesar_pointer` and `caesar_modulus` that returns a natural number in the range 0..`caesar_modulus` − 1.

If the actual value of the formal parameter `CAESAR_HASH` is `NULL`, it is replaced by a pointer to a default hash function that depends on the value of `CAESAR_BASE_AREA` and is determined according to the rules specified for function `CAESAR_USE_HASH_FUNCTION_AREA_1()` of the "area˙1" library.

The actual value of the formal parameter `CAESAR_PRINT` will be stored and associated to the cache pointed to by `*CAESAR_C`. It will be used subsequently to print the items of this cache.

Precisely, the actual value of `CAESAR_PRINT` should be a pointer to a printing procedure with two parameters `caesar_file` and `caesar_item` that prints to file `caesar_file` information about the contents (base field and/or mark field, if any) of the item pointed to by `caesar_item`.

If the actual value of the formal parameter `CAESAR_PRINT` is `NULL`, it is replaced by a pointer to a default procedure that prints the base field and the mark field, if any. The printing procedure used for the base field (respectively, the mark field) depends on the value of `CAESAR_BASE_AREA` (resp. `CAESAR_MARK_AREA`) and is determined according to the rules specified for function `CAESAR_USE_PRINT_FUNCTION_AREA_1()` of the "area˙1" library.

The actual value of the formal parameter `CAESAR_CLEANUP` will be stored and associated to the cache pointed to by `*CAESAR_C`. It will be used subsequently to clean up the contents of the items deleted from this cache.

Precisely, the actual value of `CAESAR_CLEANUP` should be a pointer to a procedure with one parameter `caesar_item` that cleans up the contents (base field and mark field, if any) of the item pointed to by `caesar_item`. This cleanup operation is useful if the base and/or the mark fields (if any) of items contain pointers to dynamic data structures (e.g., lists, sets, etc.) that must be freed when those items are deleted from the cache. For example, if the base field is a list of edges of type `CAESAR_TYPE_EDGE` that must not be kept in memory when items are deleted from the cache, a good candidate for the `CAESAR_CLEANUP` parameter is the `CAESAR_DELETE_EDGE_LIST()` procedure of the "edge" library, which deletes the list pointed to by the base field of the item pointed to by `caesar_item`, and then sets this base field to `NULL`.

If the actual value of the formal parameter `CAESAR_CLEANUP` is `NULL`, there will be no cleanup operation performed on the contents of the items when these items are deleted from the cache.

The actual value of the formal parameter `CAESAR_INFO` will be stored and associated to the cache pointed to by `*CAESAR_C`. This value should be a pointer to some data structure containing user-defined information that will be associated to this cache. The value of this pointer remains unchanged during the lifetime of the cache and can be inspected using the `CAESAR_INFO_CACHE_1()` function (see below).

. . . . . .

```
CAESAR_TYPE_CACHE_1 CAESAR_CURRENT_CACHE_1 ()
   { ... }
```

This function returns a pointer to the cache which is currently in use. It should be called only within the functions and procedures given as actual values for the formal parameters `CAESAR_SUBCACHE_SIZE`, `CAESAR_SUBCACHE_PERCENTAGE`, `CAESAR_COMPARE`, `CAESAR_HASH`, `CAESAR_PRINT`, and `CAESAR_CLEANUP` of procedure `CAESAR_CREATE_CACHE_1()` (see above); in this case, the result is a pointer to the cache created by the call to `CAESAR_CREATE_CACHE_1()`. If this function is called anywhere else in the application program, the result is undefined.

Note: This function allows to identify the cache to which the items passed as arguments to the six aforementioned functions and procedures belong, and thus to handle these items accordingly (the size and the contents of items belonging to different caches may differ). It is especially useful when the number of caches is unknown statically (e.g., when new caches are created dynamically during the execution of the application program).

.....

```
CAESAR_TYPE_NATURAL CAESAR_CURRENT_SUBCACHE_CACHE_1 ()
   { ... }
```

This function returns the index of the subcache which is currently in use; this subcache is in turn contained in the cache which is currently in use, pointed to by the result of function `CAESAR_CURRENT_CACHE_1()` (see above). It should be called only within the functions and procedures given as actual values for the formal parameters `CAESAR_SUBCACHE_ORDER`, `CAESAR_COMPARE`, `CAESAR_HASH`, `CAESAR_PRINT`, and `CAESAR_CLEANUP` of procedure `CAESAR_CREATE_CACHE_1()` (see above); in this case, the result is the index of the subcache, i.e., a natural number in the range $0..N-1$, where N is the number of subcaches in the cache created by the call to `CAESAR_CREATE_CACHE_1()`. If this function is called anywhere else in the application program, the result is undefined.

Note: This function allows to identify the subcache to which the items passed as arguments to the five aforementioned functions and procedures belong, and thus to handle these items accordingly.

.....

```
void CAESAR_DELETE_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 *CAESAR_C;
   { ... }
```

This procedure frees the memory space corresponding to the cache pointed to by `*CAESAR_C` using `CAESAR_DELETE()`. All the items currently present in the cache are freed by invoking first the cleanup function (if any) associated to the cache, and then `CAESAR_DELETE()`. Afterwards, the `NULL` value is assigned to `*CAESAR_C`.

.....

```
void CAESAR_PURGE_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
```

```
   { ... }
```

This procedure reinitializes the information associated to the subcache of index `CAESAR_N` of the cache pointed to by `CAESAR_C`. All the items currently present in the subcache are freed by invoking first the cleanup function (if any) associated to the cache, and then `CAESAR_DELETE()`. Afterwards, the subcache is exactly in the same state as after the creation of the cache using `CAESAR_CREATE_CACHE_1()`.

If the subcache index `CAESAR_N` is outside the range $0..N-1$ (where N is the number of subcaches in the cache), the effect is undefined.

.....

```
void CAESAR_PURGE_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This procedure reinitializes the information associated to the cache pointed to by `CAESAR_C`. All the items currently present in the cache are freed by invoking first the cleanup function (if any) associated to the cache, and then `CAESAR_DELETE()`. Afterwards, the cache is exactly in the same state as after its creation using `CAESAR_CREATE_CACHE_1()`.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_SEARCH_CACHE_1 (CAESAR_C, CAESAR_B, CAESAR_N, CAESAR_P)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B;
   CAESAR_TYPE_NATURAL *CAESAR_N;
   CAESAR_TYPE_POINTER *CAESAR_P;
   { ... }
```

This function determines if there exists, in the cache pointed to by `CAESAR_C`, an item whose base field is equal to the byte string pointed to by `CAESAR_B`. Byte string comparisons are performed using the comparison function associated to the cache. The search is done using the hash-function and hash-table associated to the cache.

If so, this function returns `CAESAR_TRUE`. In this case, the index of the subcache containing the existing item and the address of the item are respectively assigned to `*CAESAR_N` and `*CAESAR_P`. The number of searches and hits at the cache and the current global date of the cache are incremented. The number of hits at the subcache is incremented and the current local date of the subcache is set to the current global date of the cache. The number of hits at the item is also incremented and the date of the last access to the item is set to the current global date of the cache (the item becomes the most recently accessed item in the cache).

If not, this function returns `CAESAR_FALSE`. In this case, both variables `*CAESAR_N` and `*CAESAR_P` are left unchanged. The number of searches in the cache is incremented. The number of hits at the cache and the current global date of the cache are left unchanged.

.....

```
CAESAR_TYPE_POINTER CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)
```

```
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns a pointer to the base field of the next item to be put into the cache pointed to by `CAESAR_C`.

The base field pointed to by the result of `CAESAR_PUT_BASE_CACHE_1()` is initially undefined and must be assigned before calling some other functions of the "cache_1" library (see below).

.....

```
CAESAR_TYPE_POINTER CAESAR_PUT_MARK_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns a pointer to the mark field of the next item to be put into the cache pointed to by `CAESAR_C`. If there are no mark fields in the cache (due to the initialization parameters supplied to `CAESAR_CREATE_CACHE_1()`) the result is undefined.

The mark field pointed to by the result of `CAESAR_PUT_MARK_CACHE_1()` is always initialized to a bit string of 0's. It can be either consulted or modified.

.....

```
void CAESAR_PUT_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This procedure puts into the cache pointed to by `CAESAR_C` the item whose base field is pointed to by `CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)` and whose mark field (if any) is pointed to by `CAESAR_PUT_MARK_CACHE_1 (CAESAR_C)`.

The base field must have been assigned before this procedure is called.

This procedure also sets a field of type `CAESAR_TYPE_ERROR_CACHE_1` associated to the cache, indicating whether the put operation was carried out successfully or not; this field can be inspected using the function `CAESAR_STATUS_PUT_CACHE_1()` (see below).

The hash-table associated to the cache is updated to take into account the new item. To compute the hash-value for the base field, the hash-function associated to the cache is used.

If the put operation causes another item E contained in (some leaf subcache of) the cache to be replaced, this item is stored temporarily in a field associated to the cache until a future call to `CAESAR_PUT_CACHE_1()` or `CAESAR_SEARCH_AND_PUT_CACHE_1()` will cause another replacement to take place. Meanwhile, the item E can be inspected by using the `CAESAR_LAST_ITEM_REPLACED_CACHE_1()` procedure (see below). When the next replacement takes place, if the item E has not been inspected meanwhile by a call to `CAESAR_LAST_ITEM_REPLACED_CACHE_1()`, it will be first cleaned up using the cleanup function (if any) associated to the cache, and then freed using `CAESAR_DELETE()`; otherwise, the item E will not be cleaned up (because it is the user's responsibility to manage the memory possibly referenced in the contents of this item), but freed using `CAESAR_DELETE()` only.

Note: the cache is implemented in such a way that if a memory shortage occurs during a put operation when the cache is not already full, all the subcaches are considered to become full and their associated

replacement strategies start to be used.

. . . . . .

```
CAESAR_TYPE_BOOLEAN CAESAR_SEARCH_AND_PUT_CACHE_1 (CAESAR_C, CAESAR_N, CAESAR_P)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL *CAESAR_N;
   CAESAR_TYPE_POINTER *CAESAR_P;
   { ... }
```

This function is a combination of the function `CAESAR_SEARCH_CACHE_1()` and the procedure `CAESAR_PUT_CACHE_1()` defined above.

The base field pointed to by `CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)` must have been assigned before this function is called.

It first determines if there exists, in the cache pointed to by `CAESAR_C`, an item whose base field is equal to the base field of the item pointed to by `CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)`. Byte string comparisons are performed using the comparison function associated to the cache. The search is done using the hash-function and hash-table associated to the cache.

If so, this function returns `CAESAR_TRUE`. In this case, the index of the subcache containing the existing item and the address of the item are respectively assigned to `*CAESAR_N` and `*CAESAR_P`. The number of searches and hits at the cache and the current global date of the cache are incremented. The number of hits at the subcache is incremented and the current local date of the subcache is set to the current global date of the cache. The number of hits at the item is also incremented and the date of the last access to the item is set to the current global date of the cache (the item becomes the most recently accessed item in the cache). The field of type `CAESAR_TYPE_ERROR_CACHE_1` associated to the cache is set to `CAESAR_NONE_CACHE_1`.

If not, this function returns `CAESAR_FALSE`. In this case, it puts into the cache pointed to by `CAESAR_C` the item whose base field is pointed to by `CAESAR_PUT_BASE_CACHE_1 (CAESAR_C)` and whose mark field (if any) is pointed to by `CAESAR_PUT_MARK_CACHE_1 (CAESAR_C)`. The hash-table associated to the cache is updated to take into account the new item. The number of searches in the cache and the current global date of the cache are incremented.

The field of type `CAESAR_TYPE_ERROR_CACHE_1` associated to the cache is set to indicate whether the put operation was carried out successfully or not. If the put operation succeeded, variable `*CAESAR_N` is assigned the value 0 (since the item was put into the root subcache, of index 0) and variable `*CAESAR_P` is assigned the address of the item, which is now contained in the cache. If the put operation failed (because a cycle was detected in the parent relation between subcaches), the variables `*CAESAR_N` and `*CAESAR_P` are left unchanged.

If the put operation causes another item E contained in (some leaf subcache of) the cache to be replaced, this item is stored temporarily in a field associated to the cache until a future call to `CAESAR_PUT_CACHE_1()` or `CAESAR_SEARCH_AND_PUT_CACHE_1()` will cause another replacement to take place. Meanwhile, the item E can be inspected by using the `CAESAR_LAST_ITEM_REPLACED_CACHE_1()` procedure (see below). When the next replacement takes place, if the item E has not been inspected meanwhile by a call to `CAESAR_LAST_ITEM_REPLACED_CACHE_1()`, it will be first cleaned up using the cleanup function (if any) associated to the cache, and then freed using `CAESAR_DELETE()`; otherwise, the item E will not be cleaned up (because it is the user's responsibility to manage the memory possibly referenced in the contents of this item), but freed using `CAESAR_DELETE()` only.

Note: the cache is implemented in such a way that if a memory shortage occurs during a put operation when the cache is not already full, all the subcaches are considered to become full and their associated replacement strategies start to be used.

. . . . . .

```
CAESAR_TYPE_ERROR_CACHE_1 CAESAR_STATUS_PUT_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the status of the last put operation performed by a call to the procedure `CAESAR_PUT_CACHE_1()` or to the function `CAESAR_SEARCH_AND_PUT_CACHE_1()` (see above) on the cache pointed to by `CAESAR_C`.

. . . . . .

```
CAESAR_TYPE_POINTER CAESAR_MINIMAL_ITEM_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns the address of the smallest item contained in the subcache of index `CAESAR_N` of the cache pointed to by `CAESAR_C`. The smallest item is determined according to the order relation underlying the replacement strategy of the subcache of index `CAESAR_N`.

If the index `CAESAR_N` is outside the range $0..N-1$ (where N is the number of subcaches in the cache) or the subcache of index `CAESAR_N` is empty, the result is undefined.

. . . . . .

```
void CAESAR_DELETE_ITEM_CACHE_1 (CAESAR_C, CAESAR_N, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   CAESAR_TYPE_POINTER *CAESAR_B;
   { ... }
```

This procedure deletes, for the subcache of index `CAESAR_N` of the cache pointed to by `CAESAR_C`, the item whose base field is pointed to by `*CAESAR_B`. The item is freed by invoking first the cleanup function (if any) associated to the cache, and then `CAESAR_DELETE()`. Afterwards, the `NULL` value is assigned to `*CAESAR_B`.

The number of items in the subcache of index `CAESAR_N` and in the cache is decremented. The current global date of the cache is incremented and the current local date of the subcache of index `CAESAR_N` is set to the current global date of the cache.

If no item stored in the subcache of index `CAESAR_N` of the cache has a base field at address `CAESAR_B`, the effect is undefined.

If the index `CAESAR_N` is outside the range $0..N-1$ (where N is the number of subcaches in the cache) or the subcache of index `CAESAR_N` is empty, the effect is undefined.

```
void CAESAR_LAST_ITEM_REPLACED_CACHE_1 (CAESAR_C, CAESAR_N, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL *CAESAR_N;
   CAESAR_TYPE_POINTER *CAESAR_B;
   { ... }
```

This procedure respectively assigns to the variables `*CAESAR_N` and `*CAESAR_B` the index of the subcache and the address of the item that was replaced in that subcache when the last call to `CAESAR_PUT_CACHE_1()` or to `CAESAR_SEARCH_AND_PUT_CACHE_1()` was performed on the cache pointed to by `CAESAR_C` and caused a replacement to take place.

This procedure also sets an internal field of the cache indicating that the last item replaced was inspected, and from now on it is the user's responsibility to manage the memory possibly referenced in the contents of the item pointed to by `*CAESAR_B`. Thus, when some future call to `CAESAR_PUT_CACHE_1()` or to `CAESAR_SEARCH_AND_PUT_CACHE_1()` on the cache will cause another item to be replaced, the item pointed to by `*CAESAR_B` will not be cleaned up by invoking the cleanup function (if any) associated to the cache, but its contents will be freed by invoking `CAESAR_DELETE()` only.

If none of the previous calls to `CAESAR_PUT_CACHE_1()` or to `CAESAR_SEARCH_AND_PUT_CACHE_1()` caused a replacement to take place, then the values N (where N is the number of subcaches in the cache) and `NULL` are respectively assigned to the variables `*CAESAR_N` and `*CAESAR_B`.

Note: The item pointed to by `*CAESAR_B` can be handled in the same way as an ordinary item present in the cache, e.g., the address of its mark field can be retrieved using the `CAESAR_RETRIEVE_B_M_CACHE_1()` procedure.

```
void CAESAR_UPDATE_CACHE_1 (CAESAR_C, CAESAR_N, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   CAESAR_TYPE_POINTER CAESAR_B;
   { ... }
```

This procedure simulates, for the subcache of index `CAESAR_N` of the cache pointed to by `CAESAR_C`, a hit at the item whose base field is pointed to by `CAESAR_B`.

The current global date of the cache is incremented and the current local date of the subcache of index `CAESAR_N` is set to the current global date of the cache. The date of the last access to the item pointed to by `CAESAR_B` is set to the current global date of the cache (the item becomes the most recently accessed item in the cache). The number of hits at the item is incremented. Finally, the subcache is updated depending whether the item has become smaller or greater according to the order relation underlying the replacement strategy associated to the subcache.

If no item stored in the subcache of index `CAESAR_N` of the cache has a base field at address `CAESAR_B`, the effect is undefined.

If the subcache index `CAESAR_N` is outside the range $0..N-1$ (where N is the number of subcaches in the cache), the effect is undefined.

Note: If the items of the cache contain mark fields, and if the subcache is equipped with a user-defined replacement strategy whose underlying order relation depends on the contents of mark fields, this procedure should be called after any modification of the mark field of an item of the subcache in order to bring the subcache to a consistent state.

.....

```
CAESAR_TYPE_NATURAL CAESAR_CURRENT_DATE_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns the date of the last modifying operation performed on an item present in the subcache of index `CAESAR_N` of the cache pointed to by `CAESAR_C`.

If the subcache index `CAESAR_N` is outside the range $0..N-1$ (where N is the number of subcaches in the cache), the result is undefined.

.....

```
CAESAR_TYPE_NATURAL CAESAR_CURRENT_DATE_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the current global date of the cache pointed to by `CAESAR_C`.

.....

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_ITEMS_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns the number of items currently contained in the subcache of index `CAESAR_N` of the cache pointed to by `CAESAR_C`.

If the subcache index `CAESAR_N` is outside the range $0..N-1$ (where N is the number of subcaches in the cache), the result is undefined.

.....

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_ITEMS_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the number of items currently contained in the cache pointed to by `CAESAR_C`.

.....

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_SEARCHES_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the number of search operations performed on the cache pointed to by `CAESAR_C`.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_HITS_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns the number of hits at the subcache of index `CAESAR_N` of the cache pointed to by `CAESAR_C`.

If the subcache index `CAESAR_N` is outside the range $0..N-1$ (where N is the number of subcaches in the cache), the result is undefined.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_HITS_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the number of hits at the cache pointed to by `CAESAR_C`.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_ITEM_PUT_DATE_CACHE_1 (CAESAR_C, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B;
   { ... }
```

This function returns the date when the item whose base field is pointed to by `CAESAR_B` was put into the cache pointed to by `CAESAR_C`.

If no item stored in (some subcache of) the cache has a base field at address `CAESAR_B`, the result is undefined.

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_ITEM_CURRENT_DATE_CACHE_1 (CAESAR_C, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B;
   { ... }
```

This function returns, for the cache pointed to by CAESAR_C, the date of the last access to the item whose base field is pointed to by CAESAR_B.

If no item stored in (some subcache of) the cache has a base field at address CAESAR_B, the result is undefined.

.....

```
CAESAR_TYPE_NATURAL CAESAR_ITEM_NUMBER_OF_HITS_CACHE_1 (CAESAR_C, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B;
   { ... }
```

This function returns, for the cache pointed to by CAESAR_C, the number of hits at the item whose base field is pointed to by CAESAR_B.

If no item stored in (some subcache of) the cache has a base field at address CAESAR_B, the result is undefined.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_EMPTY_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns a value different from 0 if the subcache of index CAESAR_N of the cache pointed to by CAESAR_C is empty, and 0 otherwise. CAESAR_EMPTY_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N) is always equivalent to:

```
CAESAR_NUMBER_OF_ITEMS_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N) == 0
```

If the subcache index CAESAR_N is outside the range $0..N-1$ (where N is the number of subcaches in the cache), the result is undefined.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_EMPTY_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns a value different from 0 if the cache pointed to by CAESAR_C is empty, and 0 otherwise. CAESAR_EMPTY_CACHE_1 (CAESAR_C) is always equivalent to:

```
CAESAR_NUMBER_OF_ITEMS_CACHE_1 (CAESAR_C) == 0
```

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_FULL_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_NATURAL CAESAR_N;
   { ... }
```

This function returns a value different from 0 if the subcache of index `CAESAR_N` of the cache pointed to by `CAESAR_C` is full, and 0 otherwise. `CAESAR_FULL_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)` is always equivalent to:

`CAESAR_NUMBER_OF_ITEMS_SUBCACHE_CACHE_1 (CAESAR_C, CAESAR_N)` == K

where K denotes the maximum number of items that the subcache can contain.

If the subcache index `CAESAR_N` is outside the range $0..N-1$ (where N is the number of subcaches in the cache), the result is undefined.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_FULL_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns a value different from 0 if the cache pointed to by `CAESAR_C` is full, and 0 otherwise. `CAESAR_FULL_CACHE_1 (CAESAR_C)` is always equivalent to:

`CAESAR_NUMBER_OF_ITEMS_CACHE_1 (CAESAR_C)` == K

where K denotes the maximum number of items that the cache can contain.

.....

```
CAESAR_TYPE_POINTER CAESAR_INFO_CACHE_1 (CAESAR_C)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This function returns the pointer to the user information associated to the cache pointed to by `CAESAR_C` when this cache was created using `CAESAR_CREATE_CACHE_1()`. Precisely, the result returned by this function is the value of the formal parameter `CAESAR_INFO` supplied at the call to `CAESAR_CREATE_CACHE_1()`.

.....

```
void CAESAR_RETRIEVE_B_M_CACHE_1 (CAESAR_C, CAESAR_B, CAESAR_M)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_B;
   CAESAR_TYPE_POINTER *CAESAR_M;
   { ... }
```

This procedure computes, for the cache pointed to by `CAESAR_C`, the address of the mark field of the item whose base field is pointed to by `CAESAR_B`. This address is assigned to `*CAESAR_M`.

If no item stored in (some subcache of) the cache has a base field at address `CAESAR_B`, the effect is undefined.

If there are no mark fields in the cache (due to the initialization parameters supplied to `CAESAR_CREATE_CACHE_1()`), the effect is undefined.

.....

```
void CAESAR_RETRIEVE_M_B_CACHE_1 (CAESAR_C, CAESAR_M, CAESAR_B)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_POINTER CAESAR_M;
   CAESAR_TYPE_POINTER *CAESAR_B;
   { ... }
```

This procedure computes, for the cache pointed to by `CAESAR_C`, the address of the base field of the item whose mark field is pointed to by `CAESAR_M`. This address is assigned to `*CAESAR_B`.

If no item stored in (some subcache of) the cache has a mark field at address `CAESAR_M`, the effect is undefined.

If there are no mark fields in the cache (due to the initialization parameters supplied to `CAESAR_CREATE_CACHE_1()`), the effect is undefined.

.....

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_CACHE_1 (CAESAR_C, CAESAR_FORMAT)
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the cache pointed to by `CAESAR_C` will be printed by the procedure `CAESAR_PRINT_CACHE_1()` (see below). Currently, the following formats are available:

- With format 0, statistical information concerning the cache is displayed such as: the number of items, the replacement strategy and the number of hits for each subcache of the cache, the total number of searches and hits for the whole cache, etc.
- (no other format available yet)

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

When called with `CAESAR_FORMAT` between 0 and 0, this fonction sets the current format of `CAESAR_C` 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_C` 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_C`. If `CAESAR_FORMAT` is equal to the constant `CAESAR_MAXIMAL_FORMAT`, the result is the maximal format value (i.e., 0). In all other cases, the effect of this function is undefined.

.....

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_CACHE_1 ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as it might be removed from future versions of the OPEN/CÆSAR. Use function `CAESAR_FORMAT_CACHE_1()` instead, called with argument `CAESAR_MAXIMAL_FORMAT`.

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

.....

```
void CAESAR_PRINT_CACHE_1 (CAESAR_FILE, CAESAR_C)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_CACHE_1 CAESAR_C;
   { ... }
```

This procedure prints on file `CAESAR_FILE` an ASCII text containing various informations about the cache pointed to by `CAESAR_C`. The nature of these informations is determined by the current format of the cache pointed to by `CAESAR_C`.

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

.....

Chapter 13

The "diagnostic_1" library (version 1.1)

by Hubert Garavel

13.1 Purpose

The "diagnostic_1" library provides primitives for managing diagnostics.

13.2 Usage

The "diagnostic_1" library consists of:

- a predefined header file "`caesar_diagnostic_1.h`";
- the precompiled library file "`libcaesar.a`", which implements the features described in "`caesar_diagnostic_1.h`".

Note: The "diagnostic_1" library is a software layer built above the primitives offered by the "standard", "edge" and "stack_1" libraries, and by the OPEN/CÆSAR graph module.

Note: The "diagnostic_1" library relies on the "edge" and "stack_1" libraries. Therefore, when using the "diagnostic_1" library, there are restrictions concerning the use of the "edge" and "stack_1" library primitives. These restrictions are listed in the sequel.

13.3 Description

A "diagnostic" is an execution sequence, i.e., a list of states and transitions starting from the initial state of the graph and leading to a given state.

The "size" of a diagnostic is defined to be the number of states (not the number of transitions) in the corresponding execution sequence. This definition is compatible with the convention used for function `CAESAR_DEPTH_STACK_1()`.

Diagnostics are to be used during depth-first graph exploration. For this reason, this library uses (and is compatible with) stacks provided by the "stack_1" library. Let's consider, for instance, an

OPEN/CÆSAR user program that searches for deadlocks. Every time a deadlock state is detected, the appropriate diagnostic is the contents of the stack, i.e., the execution sequence leading from the initial state to the current deadlock state.

The "diagnostic_1" allows to control the way diagnostics are printed. For instance, it allows to display only the shortest diagnostic sequence detected. Other "strategies" are also available. The data structure used to store diagnostics and related information is called a "diagnostic structure".

13.4 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_DIAGNOSTIC_1;
```

This type denotes a pointer to the concrete representation of a diagnostic structure, which is supposed to be "opaque".

.....

```
typedef enum {
     CAESAR_NONE_DIAGNOSTIC_1,
     CAESAR_ALL_DIAGNOSTIC_1,
     CAESAR_FIRST_DIAGNOSTIC_1,
     CAESAR_DECREASING_DIAGNOSTIC_1,
     CAESAR_SHORTEST_DIAGNOSTIC_1
}    CAESAR_TYPE_STRATEGY_DIAGNOSTIC_1;
```

This enumerated type defines the "strategy" used to print diagnostics during graph exploration. Two steps are to be distinguished: every time an execution sequence of interest is detected, it is "recorded" using the `CAESAR_RECORD_DIAGNOSTIC_1()` function defined below. When the depth-first exploration is completed, a "summary request" is performed using the `CAESAR_SUMMARIZE_DIAGNOSTIC_1()` function defined below. The effect of these two functions depends on the chosen strategy:

- With `CAESAR_NONE_DIAGNOSTIC_1`, only the result of the exploration is printed (i.e., some diagnostic has been found or not). Diagnostics themselves are not printed.
- With `CAESAR_ALL_DIAGNOSTIC_1`, all diagnostics are printed when they are recorded.
- With `CAESAR_FIRST_DIAGNOSTIC_1`, only the first diagnostic is printed when it is recorded; the next ones will not be printed.
- With `CAESAR_DECREASING_DIAGNOSTIC_1`, the first diagnostic is printed when it is recorded; the next ones will be printed iff their size is strictly smaller than all previously recorded diagnostics.
- With `CAESAR_SHORTEST_DIAGNOSTIC_1`, diagnostics are not printed when they are recorded, but the diagnostic with the smallest size is stored in a diagnostic structure. This shortest diagnostic (if any) will be printed when a summary request is emitted.

.....

```
void CAESAR_CREATE_DIAGNOSTIC_1 (CAESAR_D, CAESAR_STRATEGY,
                                 CAESAR_DEPTH, CAESAR_FILE,
                                 CAESAR_PROGRESS, CAESAR_HEADER,
                                 CAESAR_FOOTER, CAESAR_SEPARATOR,
                                 CAESAR_REPORT)
   CAESAR_TYPE_DIAGNOSTIC_1 *CAESAR_D;
   CAESAR_TYPE_STRATEGY_DIAGNOSTIC_1 CAESAR_STRATEGY;
   CAESAR_TYPE_NATURAL CAESAR_DEPTH;
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_STRING CAESAR_PROGRESS;
   CAESAR_TYPE_STRING CAESAR_HEADER;
   CAESAR_TYPE_STRING CAESAR_FOOTER;
   CAESAR_TYPE_STRING CAESAR_SEPARATOR;
   CAESAR_TYPE_STRING CAESAR_REPORT;
   { ... }
```

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

Note: because `CAESAR_TYPE_DIAGNOSTIC_1` is a pointer type, any variable `CAESAR_D` of type `CAESAR_TYPE_DIAGNOSTIC_1` must be allocated before used, for instance using:

```
CAESAR_CREATE_DIAGNOSTIC_1 (&CAESAR_D, ...);
```

Note: Before calling this procedure, the `CAESAR_INIT_STACK_1()` procedure of the "stack_1" library should have been invoked (because the "diagnostic_1" library uses the "stack_1" library). Also, the restrictions concerning the `CAESAR_INIT_STACK_1()` procedure should be observed.

The actual values of the remaining formal parameters will be stored and associated to the diagnostic structure pointed to by `*CAESAR_D`.

The value of `CAESAR_STRATEGY` determines the strategy used for this diagnostic structure.

The value of `CAESAR_DEPTH` determines the "maximal depth" used for this diagnostic structure, i.e., an upper bound on the sizes of the diagnostics. If `CAESAR_DEPTH` is equal to zero, there is no upper bound.

The value of `CAESAR_FILE` determines the output file to which the diagnostics will be printed. Before any diagnostic is printed, `CAESAR_FILE` must have been properly opened, for instance using `fopen(3)`.

The values of `CAESAR_PROGRESS`, `CAESAR_HEADER`, `CAESAR_FOOTER`, `CAESAR_SEPARATOR`, and `CAESAR_REPORT` are character strings to be used for printing the diagnostics. The exact semantics of these parameters will not be defined here. For interoperability with the other tools of the CADP toolset (see the "`seq`" manual page for a definition of the SEQ format), it is advised to use the following actual values:

```
   CAESAR_PROGRESS  = "*** sequence(s) found at depth "
   CAESAR_HEADER    = "*** sequence found at depth %u\n\n\001<initial state>\002\n"
   CAESAR_FOOTER    = "\001<goal state>\002\n\n"
   CAESAR_SEPARATOR = "[]\n\n"
   CAESAR_REPORT    = "*** no sequence found\n\n"
```

or, depending on the context:

```
    CAESAR_PROGRESS  = "*** deadlock(s) found at depth "
    CAESAR_HEADER    = "*** deadlock found at depth %u\n\n\001<initial state>\002\n"
    CAESAR_FOOTER    = "<deadlock>\n\n"
```

```
    CAESAR_SEPARATOR = "[]\n\n"
    CAESAR_REPORT    = "*** no deadlock found\n\n"
```

.....

```
void CAESAR_DELETE_DIAGNOSTIC_1 (CAESAR_D)
   CAESAR_TYPE_DIAGNOSTIC_1 *CAESAR_D;
   { ... }
```

This procedure frees the memory space corresponding to the diagnostic structure pointed to by `*CAESAR_D` using `CAESAR_DELETE()`. Afterwards, the `NULL` value is assigned to `*CAESAR_D`.

.....

```
void CAESAR_RECORD_DIAGNOSTIC_1 (CAESAR_D, CAESAR_K)
   CAESAR_TYPE_DIAGNOSTIC_1 CAESAR_D;
   CAESAR_TYPE_STACK_1 CAESAR_K;
   { ... }
```

This procedure records into the diagnostic structure pointed to by `*CAESAR_D` the diagnostic contained in the stack pointed to by `*CAESAR_K`. The stack pointed to by `*CAESAR_K` should contain at least one item (the initial state of the sequence); if it is empty, the result is undefined. The effects of this procedure depend on the strategy and maximal depth associated with the diagnostic structure:

- With `CAESAR_NONE_DIAGNOSTIC_1`, the diagnostic is not printed but stored into the diagnostic structure.
- With `CAESAR_ALL_DIAGNOSTIC_1`, the diagnostic is immediately printed.
- With `CAESAR_FIRST_DIAGNOSTIC_1`, the diagnostic is immediately printed if it is the first one to be recorded; otherwise nothing is done.
- With `CAESAR_DECREASING_DIAGNOSTIC_1`, the diagnostic is immediately printed if its size is less or equal than the maximal depth (if the maximal depth is not equal to zero) and strictly less than the sizes of all previously recorded diagnostics.
- With `CAESAR_SHORTEST_DIAGNOSTIC_1`, the diagnostic is not printed but stored into the diagnostic structure if its size os less or equal than the maximal depth (if the maximal depth is not equal to zero) and strictly less than the sizes of all previously recorded diagnostics. If the character string `CAESAR_PROGRESS` specified for `*CAESAR_D` is not the `NULL` pointer, it is printed, together with the size of the recorded diagnostic.

.....

```
void CAESAR_SUMMARIZE_DIAGNOSTIC_1 (CAESAR_D)
   CAESAR_TYPE_DIAGNOSTIC_1 CAESAR_D;
   { ... }
```

This procedure summarizes the current status of the diagnostic structure pointed to by `*CAESAR_D`. The effects of this procedure depends on the strategy associated with the diagnostic structure:

- With CAESAR_NONE_DIAGNOSTIC_1, the result of the exploration is displayed: if diagnostics have been recorded, the size of the most recently recorded one is printed; otherwise, the character string CAESAR_REPORT is printed if no diagnostic has been recorded into the diagnostic structure.
- With CAESAR_ALL_DIAGNOSTIC_1, the character string CAESAR_REPORT is printed if no diagnostic has been previously recorded into the diagnostic structure.
- With CAESAR_FIRST_DIAGNOSTIC_1, same as for CAESAR_ALL_DIAGNOSTIC_1.
- With CAESAR_DECREASING_DIAGNOSTIC_1, same as for CAESAR_ALL_DIAGNOSTIC_1.
- With CAESAR_SHORTEST_DIAGNOSTIC_1, the shortest diagnostic recorded is printed, or the character string CAESAR_REPORT is printed if no diagnostic has been recorded into the diagnostic structure.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_EMPTY_DIAGNOSTIC_1 (CAESAR_D)
   CAESAR_TYPE_DIAGNOSTIC_1 CAESAR_D;
   { ... }
```

This function returns the boolean value CAESAR_TRUE iff at least one diagnostic has been previously recorded (using the CAESAR_RECORD_DIAGNOSTIC_1() function) into the diagnostic structure pointed to by *CAESAR_D.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_BACKTRACK_DIAGNOSTIC_1 (CAESAR_D, CAESAR_DEPTH)
   CAESAR_TYPE_DIAGNOSTIC_1 CAESAR_D;
   CAESAR_TYPE_NATURAL CAESAR_DEPTH;
   { ... }
```

This function returns the boolean value CAESAR_TRUE iff the exploration should stop and backtrack when reaching depth CAESAR_DEPTH. For instance, in a depth-first search, the actual value of CAESAR_DEPTH should be equal to the current depth of the stack. The result depends on the strategy and maximal depth associated with the diagnostic structure pointed to by *CAESAR_D:

- With CAESAR_NONE_DIAGNOSTIC_1, the result is CAESAR_TRUE iff CAESAR_DEPTH is strictly greater than the maximal depth (if the maximal depth is not equal to zero).
- With CAESAR_ALL_DIAGNOSTIC_1, same as for CAESAR_NONE_DIAGNOSTIC_1.
- With CAESAR_FIRST_DIAGNOSTIC_1, the result is CAESAR_TRUE iff CAESAR_DEPTH is strictly greater than the maximal depth (if the maximal depth is not equal to zero) or if some diagnostic has been previously recorded into the diagnostic structure.
- With CAESAR_DECREASING_DIAGNOSTIC_1, the result is CAESAR_TRUE iff CAESAR_DEPTH is strictly greater than the maximal depth (if the maximal depth is not equal to zero) or if it is greater or equal to the size of some previously recorded diagnostic.
- With CAESAR_SHORTEST_DIAGNOSTIC_1, same as for CAESAR_DECREASING_DIAGNOSTIC_1.

Chapter 14

The "hide_1" library (version 1.1)

by Hubert Garavel

14.1 Purpose

The "hide_1" library provides primitives for processing "hiding files". These files specify which labels of a graph should be hidden (i.e., renamed into the internal action "i", or tau) using a set of regular expressions.

14.2 Usage

The "hide_1" library consists of:

- a predefined header file "`caesar_hide_1.h`";
- the precompiled library file "`libcaesar.a`", which implements the features described in "`caesar_hide_1.h`".

Note: The "hide_1" library is a software layer built above the primitives offered by the "standard" library.

14.3 Hiding files

In this section, we define the format of hiding files, as they are used in the CADP toolbox. The next sections will explain how the "hide_1" library of OPEN/CÆSAR supports (and extends) this format.

A hiding file is a text file containing a set of "hiding patterns". There is no mandatory suffix (i.e., file extension) for hiding files: any file name can be used; however, it is recommended to use one of the following suffixes ".hid" (recommended) or ".hide".

The syntax of hiding files is described by the following context-free grammar:

```
<axiom> ::= <blanks> <positive-header> <blanks> \n <pattern-list>
          |  <blanks> <negative-header> <blanks> \n <pattern-list>
```

```
<positive-header> ::= hide

<negative-header> ::= hide all but

<pattern-list> ::= <empty>
                |  <pattern> <pattern-list>

<pattern> ::= <blanks> <regexp> <blanks> \n
           |  <blanks> "<regexp>" <blanks> \n
```

where:

- `\n` denotes the newline character;
- `<blanks>` is any sequence of spaces, tabulations, carriage returns, newlines, vertical tabulations, or form feeds; these characters are those recognized by the POSIX function `isspace()`; they are always skipped and ignored;
- `<empty>` denotes the empty sequence;
- `<regexp>` is a character string specifying a regular expression according to the definition given in the `regexp`(LOCAL) manual page. The `<regexp>` may be enclosed between double quotes, which will be removed and ignored.

Note: hiding files are case sensitive: upper-case and lower-case letters are considered to be different.

Note: in the `<pattern-list>`, lines that are empty or contain only blanks will be ignored.

Semantically, a hiding file behaves as a predicate that decides whether a character string (presumably representing the label of a transition) belongs to a set of patterns. Depending on the first line of the hiding file (positive or negative header), a label will be hidden (i.e., renamed into the internal action "i", often referred to as "tau") if recognized (or vice-versa). More precisely, the effect of a hiding file F is defined as follows:

- If the first line of F is equal to `"hide"`, then any character string S that matches one (or more) `<regexp>`(s) contained in F will be hidden; if S matches no `<regexp>`, it will be kept unchanged.
- If the first line of F is equal to `"hide all but"`, then any character string S that matches no `<regexp>` contained in F will be hidden; if S matches one (or more) `<regexp>`(s), it will be kept unchanged.

For instance, the following file:

```
hide
GET
"ACK !false .*"
```

will hide all character strings equal to `"GET"` or prefixed with `"ACK !false "`. Similarly, the following file:

```
hide all but
"PUT .*"
```

will hide all character strings except those prefixed with `"PUT "`.

14.4 Generalized hiding files

The above format for hiding files is the one used in the CADP toolbox for hiding labels selectively. The "hide_1" library of OPEN/CÆSAR supports this format, while providing additional flexibility, in several directions:

- The "hide_1" library does not hide labels itself: it simply implements the predicate function that determines whether a character string (representing a label) matches or not some of the `<regexp>`'s contained in a file F. On the basis of this information, the OPEN/CÆSAR programmer is free to hide the label, or perform any other action (for instance, distinguish between input and output labels).

- The "hide_1" library allows to parameterize the definition of `<positive-header>` and `<negative-header>`. These symbols can be different from `"hide"` and `"hide all but"`; for instance, they could be replaced by `"input"` and `"output"`. The values of `<positive-header>` and `<negative-header>` are determined by two regular expressions passed as parameters to function `CAESAR_CREATE_HIDE_1()` (see below).

- The "hide_1" library also allows files without `<positive-header>` nor `<negative-header>`. This special case is obtained by giving the `NULL` value to the corresponding parameters in function `CAESAR_CREATE_HIDE_1()`. In such case, the `<axiom>` of the grammar is simply defined as `<pattern-list>`. A character string S will be recognized if it matches one of the `<regexp>`'s contained in the file.

- The "hide_1" library allows three possibilities for deciding whether a character string S matches a `<regexp>` R: *total matching* (S should match R entirely), *partial matching* (S should contain a sub-string that matches R), or *gate matching* (the first word of S, should match R entirely, the remaining part of S being ignored; the first word of S is the sub-string starting at the beginning of S and ending at the first character "`!`", "`?`", "`(`", space, or tabulation, if any, or at the end of S otherwise; in the case where S is a LOTOS label, the first word of S denotes a LOTOS gate identifier). The choice between these possibilities is determined by the value of an actual parameter passed to function `CAESAR_CREATE_HIDE_1()`. The hiding files used in the CADP toolbox follow the total match semantics.

14.5 Description

The "hide_1" library allows to process one or several hiding files at the same time. Each hiding file is read, parsed and checked; if correct, its contents are stored (under a compiled form) in a data structure called "hiding object". Afterwards, the hiding file will only be handled through a pointer to its corresponding hiding object.

14.6 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_HIDE_1;
```

This type denotes a pointer to the concrete representation of a hiding object, which is supposed to be "opaque".

. . . . . .

```
void CAESAR_CREATE_HIDE_1 (CAESAR_H, CAESAR_PATHNAME, CAESAR_POSITIVE_HEADER,
                           CAESAR_NEGATIVE_HEADER, CAESAR_KIND)
   CAESAR_TYPE_HIDE_1 *CAESAR_H;
   CAESAR_TYPE_STRING CAESAR_PATHNAME;
   CAESAR_TYPE_STRING CAESAR_POSITIVE_HEADER;
   CAESAR_TYPE_STRING CAESAR_NEGATIVE_HEADER;
   CAESAR_TYPE_NATURAL CAESAR_KIND;
   { ... }
```

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

Note: because `CAESAR_TYPE_HIDE_1` is a pointer type, any variable `CAESAR_H` of type `CAESAR_TYPE_HIDE_1` must be allocated before used, for instance using:

```
CAESAR_CREATE_HIDE_1 (&CAESAR_H, ...);
```

The actual value of the formal parameter `CAESAR_PATHNAME` denotes a character string containing the file name of the hiding file. If the file name has a suffix (see above for a discussion about suffixes for hiding files), this suffix should be part of the character string `CAESAR_PATHNAME` (no suffix will be added implicitly). The hiding file referred to by `CAESAR_PATHNAME` should exist and be readable.

As a special case, if `CAESAR_PATHNAME` is equal to `NULL`, then the hiding file will be read from the standard input.

The actual value of the formal parameter `CAESAR_POSITIVE_HEADER` denotes a character string containing a regular expression according to the definition given in the manual page of the POSIX `regexp(LOCAL)` command. This regular expression specifies the `<positive-header>` that may occur at the first line of the hiding file.

The actual value of the formal parameter `CAESAR_NEGATIVE_HEADER` denotes a character string containing a regular expression according to the definition given in the manual page of the POSIX `regexp(LOCAL)` command. This regular expression specifies the `<negative-header>` that may occur at the first line of the hiding file.

As a special case, if both `CAESAR_POSITIVE_HEADER` and `CAESAR_NEGATIVE_HEADER` are equal to `NULL`, then the hiding file should have no header line. Otherwise, `CAESAR_POSITIVE_HEADER` and `CAESAR_NEGATIVE_HEADER` should both be different from `NULL`.

Note: if the regular expressions `CAESAR_POSITIVE_HEADER` and `CAESAR_NEGATIVE_HEADER` are not exclusive (i.e., there exists at least one character string that matches both regular expressions), then the effect is undefined.

The actual value of the formal parameter `CAESAR_KIND` should be equal to 0 if total matching is desired, to 1 if partial matching is desired, or to 2 if gate matching is desired (see above for a definition of these terms).

The hiding file is parsed: its `<regexp>`'s are analyzed and stored (under a compiled form) into the hiding object `*CAESAR_H`.

So doing, various error conditions may occur: the hiding file can not be open; it is empty, or the first line matches neither the header specified by `CAESAR_POSITIVE_HEADER` nor the one specified by `CAESAR_NEGATIVE_HEADER`; `CAESAR_POSITIVE_HEADER` (resp. `CAESAR_NEGATIVE_HEADER`) is not

a valid regular expression; the hiding file has syntax errors; it contains some `<regexp>` that is not a valid regular expression; etc. In such case, a detailed error message is displayed using the `CAESAR_WARNING()` procedure, and the `NULL` value is assigned to `*CAESAR_H`.

. . . . . .

```
#define CAESAR_POSITIVE_HEADER_HIDE_1 "hide"
```

This macro-definition returns the standard positive header for the hiding files used in the CADP toolbox (see above). In such case, this macro-definition should be used as an actual value for parameter `CAESAR_POSITIVE_HEADER` when invoking function `CAESAR_CREATE_HIDE_1`.

. . . . . .

```
#define CAESAR_NEGATIVE_HEADER_HIDE_1 "hide[ \t][ \t]*all[ \t][ \t]*but"
```

This macro-definition returns the standard negative header for the hiding files used in the CADP toolbox (see above). In such case, this macro-definition should be used as an actual value for parameter `CAESAR_NEGATIVE_HEADER` when invoking function `CAESAR_CREATE_HIDE_1`.

. . . . . .

```
void CAESAR_DELETE_HIDE_1 (CAESAR_H)
   CAESAR_TYPE_HIDE_1 *CAESAR_H;
   { ... }
```

This procedure frees the memory space corresponding to the hiding object pointed to by `*CAESAR_H` using `CAESAR_DELETE()`. Afterwards, the `NULL` value is assigned to `*CAESAR_H`.

. . . . . .

```
CAESAR_TYPE_BOOLEAN CAESAR_TEST_HIDE_1 (CAESAR_H, CAESAR_S)
   CAESAR_TYPE_HIDE_1 CAESAR_H;
   CAESAR_TYPE_STRING CAESAR_S;
   { ... }
```

This function returns `CAESAR_TRUE` if the character string `CAESAR_S` is recognized by the hiding object pointed to by `CAESAR_H` (according to the semantics defined above), or `CAESAR_FALSE` otherwise.

. . . . . .

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_HIDE_1 (CAESAR_H, CAESAR_FORMAT)
   CAESAR_TYPE_HIDE_1 CAESAR_H;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the hiding object pointed to by `CAESAR_H` will be printed by the procedure `CAESAR_PRINT_HIDE_1()` (see below). Currently, the following format is available:

- With format 0, information about the hiding object is displayed such as: the pathname of the corresponding hiding file, the positive header (if any), the negative header (if any), the number of patterns, the list of patterns, etc.
- (no other format available yet).

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

When called with `CAESAR_FORMAT` between 0 and 0, this fonction sets the current format of `CAESAR_H` 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_H` 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_H`. If `CAESAR_FORMAT` is equal to the constant `CAESAR_MAXIMAL_FORMAT`, the result is the maximal format value (i.e., 0). In all other cases, the effect of this function is undefined.

.....

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_HIDE_1 ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as itmight be removed from future versions of the OPEN/CÆSAR. Use function `CAESAR_FORMAT_HIDE_1()` instead, called with argument `CAESAR_MAXIMAL_FORMAT`.

This function returns the maximal format value available for printing hiding objects.

.....

```
void CAESAR_PRINT_HIDE_1 (CAESAR_FILE, CAESAR_H)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_HIDE_1 CAESAR_H;
   { ... }
```

This procedure prints to file `CAESAR_FILE` a text containing information about the hiding object pointed to by `CAESAR_H`. The nature of the information is determined by the current format of the hiding object pointed to by `CAESAR_H`.

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

.....

Chapter 15

The "rename_1" library (version 1.1)

by Hubert Garavel

15.1 Purpose

The "rename_1" library provides primitives for processing "renaming files". These files specify which labels of a graph should be renamed, and how to rename them, using a set of regular expressions.

15.2 Usage

The "rename_1" library consists of:

- a predefined header file "`caesar_rename_1.h`";
- the precompiled library file "`libcaesar.a`", which implements the features described in "`caesar_rename_1.h`".

Note: The "rename_1" library is a software layer built above the primitives offered by the "standard" library.

15.3 Renaming files

In this section, we define the format of renaming files, as they are used in the CADP toolbox. The next sections will explain how the "renaming_1" library of OPEN/CÆSAR supports (and extends) this format.

A renaming file is a text file containing a set of "renaming patterns". There is no mandatory suffix (i.e., file extension) for renaming files: any file name can be used; however, it is recommended to use one of the following suffixes ".ren" (recommended) or ".rename".

The syntax of renaming files is described by the following context-free grammar:

```
<axiom> ::= <blanks> <header> <blanks> \n <renaming-list>
```

```
<header> ::= rename

<renaming-list> ::= <empty>
                  |  <renaming> <renaming-list>

<renaming> ::= <left-pattern> -> <right-pattern> \n

<left-pattern> ::= <blanks> <left-regexp> <blanks>
                 |  <blanks> "<left-regexp>" <blanks>

<right-pattern> ::= <blanks> <right-regexp> <blanks>
                  |  <blanks> "<right-regexp>" <blanks>
```

where:

- `\n` denotes the newline character;
- `<blanks>` is any sequence of spaces, tabulations, carriage returns, newlines, vertical tabulations, or form feeds; these characters are those recognised by the POSIX function `isspace()`; they are always skipped and ignored;
- `<empty>` denotes the empty sequence;
- `<left-regexp>` is a character string specifying a regular expression according to the definition given in the `regexp`(LOCAL) manual page. The `<left-regexp>` may be enclosed between double quotes, which will be removed and ignored.
- `<right-regexp>` is a character string specifying a "replacement" expression according to the definition given in the manual page of the POSIX `regexp(LOCAL)` command. The `<right-regexp>` can be surrounded by double quotes, which will be removed and ignored.

Note: renaming files are case sensitive: upper-case and lower-case letters are considered to be different.

Note: in the `<renaming-list>`, lines that are empty or contain only blanks will be ignored.

Semantically, a renaming file behaves as function that maps a character string S (presumably representing the label of a transition) to another character string (the renamed label).

More precisely, the effect of a renaming file F on a character string S is defined as follows. The `<renaming-list>` of S is scanned from top to bottom in order to determine the first `<left-regexp>` matched by S. If this `<left-regexp>` does not exist, S is kept unchanged. If it exists, S will be renamed into the corresponding `<right-regexp>`. Renaming is performed according to the conventions for text substitution defined in the POSIX command `regexp(LOCAL)`.

Note: If S matches several `<left-regexp>`s, only the first one is taken into account. Renamings do not cumulate (although such behaviour can be explicitly programmed by invoking function `CAESAR_APPLY_RENAME_1()` several times).

For instance, the following file:

```
rename
GET -> PUT
PUT -> GET
```

will swap the labels named `"GET"` and `"PUT"`. Similarly, the following file:

```
rename
"PUT !\([A-Z]*\) !\([A-Z]*\)" -> "PUT !\2 !\1"
```

will swap the offers of `"PUT"` labels, e.g., `"PUT !ABC !XYZ"` will be renamed into `"PUT !XYZ !ABC"`.

15.4 Generalized renaming files

The above format for renaming files is the one used in the CADP toolbox for renaming labels selectively. The "rename_1" library of OPEN/CÆSAR supports this format, while providing additional flexibility, in several directions:

- The "rename_1" library allows to parameterize the definition of `<header>`, which can therefore be different from `"rename"`. The value of `<header>` is determined by a regular expression passed as parameter to function `CAESAR_CREATE_RENAME_1()` (see below).

- The "rename_1" library also allows files without `<header>`. This special case is obtained by giving the `NULL` value to the corresponding parameter in function `CAESAR_CREATE_RENAME_1()`. In such case, the `<axiom>` of the grammar is simply defined as `<renaming-list>`, which does not change the semantics.

- The "rename_1" library allows four possibilities for renaming a character string S according to a `<left-regexp>` R: *total matching* (S should match R entirely, otherwise renaming does not occur), *single partial matching* (renaming is performed for the first sub-string of S that matches R), *multiple partial matching* (renaming is performed for every sub-string of S that matches R), or *gate matching* (renaming is only performed for the first word of S and only if this first word matches R, the remaining part of S being unchanged in any case; the first word of S is the sub-string starting at the beginning of S and ending at the first character "`!`", "`?`", "`(`", space, or tabulation, if any, or at the end of S otherwise; in the case where S is a LOTOS label, the first word of S denotes a LOTOS gate identifier). The choice between these possibilities is determined by the value of an actual parameter passed to function `CAESAR_CREATE_RENAME_1()`. The renaming files used in the CADP toolbox follow the total match semantics.

15.5 Description

The "rename_1" library allows to process one or several renaming files at the same time. Each renaming file is read, parsed and checked; if correct, its contents are stored (under a compiled form) in a data structure called "renaming object". Afterwards, the renaming file will only be handled through a pointer to its corresponding renaming object.

15.6 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_RENAME_1;
```

This type denotes a pointer to the concrete representation of a renaming object, which is supposed to be "opaque".

. . . . . .

```
void CAESAR_CREATE_RENAME_1 (CAESAR_R, CAESAR_PATHNAME, CAESAR_HEADER,
                             CAESAR_KIND)
   CAESAR_TYPE_RENAME_1 *CAESAR_R;
   CAESAR_TYPE_STRING CAESAR_PATHNAME;
   CAESAR_TYPE_STRING CAESAR_HEADER;
   CAESAR_TYPE_NATURAL CAESAR_KIND;
   { ... }
```

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

Note: because `CAESAR_TYPE_RENAME_1` is a pointer type, any variable `CAESAR_R` of type `CAESAR_TYPE_RENAME_1` must be allocated before used, for instance using:

```
CAESAR_CREATE_RENAME_1 (&CAESAR_R, ...);
```

The actual value of the formal parameter `CAESAR_PATHNAME` denotes a character string containing the file name of the renaming file. If the file name has a suffix (see above for a discussion about suffixes for renaming files), this suffix should be part of the character string `CAESAR_PATHNAME` (no suffix will be added implicitly). The renaming file referred to by `CAESAR_PATHNAME` should exist and be readable.

As a special case, if `CAESAR_PATHNAME` is equal to `NULL`, then the renaming file will be read from the standard input.

The actual value of the formal parameter `CAESAR_HEADER` denotes a character string containing a regular expression according to the definition given in the manual page of the POSIX `regexp(LOCAL)` command. This regular expression specifies the `<header>` that must occur at the first line of the renaming file.

As a special case, if `CAESAR_HEADER` is equal to `NULL`, then the renaming file should have no header line.

The actual value of the formal parameter `CAESAR_KIND` should be equal to 0 if total matching is desired, to 1 if multiple partial matching is desired, to 2 if single partial matching is desired, or to 3 if gate matching is desired (see above for a definition of these terms).

The renaming file is parsed: its `<left-regexp>`'s and `<right-regexp>`'s are analyzed and stored (under a compiled form) into the renaming object `*CAESAR_R`.

So doing, various error conditions may occur: the renaming file can not be open; it is empty, or the first line does not match the header specified by `CAESAR_HEADER`; `CAESAR_HEADER` is not a valid regular expression; the renaming file has syntax errors; it contains some `<left-regexp>` (resp. some `right-regexp>` that is not a valid regular expression (resp. replacement); etc. In such case, a detailed error message is displayed using the `CAESAR_WARNING()` procedure, and the `NULL` value is assigned to `*CAESAR_R`.

.....

```
#define CAESAR_HEADER_RENAME_1 "rename"
```

This macro-definition returns the standard header for the renaming files used in the CADP toolbox (see above). In such case, this macro-definition should be used as an actual value for parameter `CAESAR_HEADER` when invoking function `CAESAR_CREATE_RENAME_1`.

```
void CAESAR_DELETE_RENAME_1 (CAESAR_R)
   CAESAR_TYPE_RENAME_1 *CAESAR_R;
   { ... }
```

This procedure frees the memory space corresponding to the renaming object pointed to by `*CAESAR_R` using `CAESAR_DELETE()`. Afterwards, the `NULL` value is assigned to `*CAESAR_R`.

```
CAESAR_TYPE_STRING CAESAR_APPLY_RENAME_1 (CAESAR_R, CAESAR_S)
   CAESAR_TYPE_RENAME_1 CAESAR_R;
   CAESAR_TYPE_STRING CAESAR_S;
   { ... }
```

This function attempts to rename the character string `CAESAR_S` according to the renaming object pointed to by `CAESAR_R`.

If renaming succeeds, this function returns a character string containing the renamed string. The address of this character string is left unspecified, but it is assumed to be different from `CAESAR_S`.

If renaming fails, this function returns `CAESAR_S`. It is therefore possible to decide whether renaming succeeded or not, by comparing the result to the second parameter passed to `CAESAR_APPLY_RENAME_1`.

Note: in any case, the contents of `CAESAR_S` will not be modified (there is no side effect).

Note: when renaming succeeds, it is not allowed to modify the character string returned by `CAESAR_APPLY_RENAME_1()` nor to free it, for instance using `free(3)`.

Note: when renaming succeeds, the contents of the character string returned by `CAESAR_APPLY_RENAME_1()` may be destroyed by a subsequent call to this function. In particular, it is forbidden to call `CAESAR_APPLY_RENAME_1()` by giving to `CAESAR_S` the value returned by a former call to `CAESAR_APPLY_RENAME_1()` for which renaming succeeded. For instance, the following call is forbidden in the general case:

```
CAESAR_APPLY_RENAME_1 (..., CAESAR_APPLY_RENAME_1 (..., ...));
```

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_RENAME_1 (CAESAR_R, CAESAR_FORMAT)
   CAESAR_TYPE_RENAME_1 CAESAR_R;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the renaming object pointed to by `CAESAR_R` will be printed by the procedure `CAESAR_PRINT_RENAME_1()` (see below). Currently, the following formats are available:

- With format 0, information about the renaming object is displayed such as: the pathname of the corresponding renaming file, the header (if any), the number of patterns, the list of left and right patterns, etc.
- (no other format available yet).

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

When called with `CAESAR_FORMAT` between 0 and 0, this fonction sets the current format of `CAESAR_R` 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_R` 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_R`. If `CAESAR_FORMAT` is equal to the constant `CAESAR_MAXIMAL_FORMAT`, the result is the maximal format value (i.e., 0). In all other cases, the effect of this function is undefined.

.....

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_RENAME_1 ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as it might be removed from future versions of the Open/Cæsar. Use function `CAESAR_FORMAT_RENAME_1()` instead, called with argument `CAESAR_MAXIMAL_FORMAT`.

This function returns the maximal format value available for printing renaming objects.

.....

```
void CAESAR_PRINT_RENAME_1 (CAESAR_FILE, CAESAR_R)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_RENAME_1 CAESAR_R;
   { ... }
```

This procedure prints to file `CAESAR_FILE` a text containing information about the renaming object pointed to by `CAESAR_R`. The nature of the information is determined by the current format of the renaming object pointed to by `CAESAR_R`.

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

.....

Chapter 16

The "mask_1" library (version 1.1)

by Nicolas Descoubes and Hubert Garavel

16.1 Purpose

The "mask_1" library provides primitives for applying sequences of hiding and renaming operations to memory areas (such as labels) converted to character strings. In particular, this allows to hide and/or rename labels on the fly.

16.2 Usage

The "mask_1" library consists of:

- a predefined header file "`caesar_mask_1.h`";
- the precompiled library file "`libcaesar.a`", which implements the features described in "`caesar_mask_1.h`".

Note: The "mask_1" library is a software layer built above the primitives offered by the "standard", "area_1", "hide_1", and "rename_1" libraries.

16.3 Description

The "mask_1" library allows to apply successive operations to memory areas (for instance, labels). These operations are, first, the conversion of a memory area to a character string, and then a sequence of hiding and/or renaming operations on the resulting character string.

Hiding a character string consists in renaming it into the internal action "i" (or tau) according to a "hiding file", which contains a list of regular expressions. The format of "hiding files" is defined in the "hide_1" library.

Renaming consists in applying string substitutions according to a "renaming file", which contains a list of substitution patterns defined using regular expressions. The format of "renaming files" is defined in the "rename_1" library.

The "mask_1" library targets at efficiency. Each hiding file and each renaming file is parsed and checked only once. In addition, a hash-based cache table is available to speed up performance; this table stores the character strings resulting from prior operations performed by the masking object, so as to avoid multiple, redundant computations.

The "mask_1" library introduces the concept of "masking object", which is an abstract data structure containing (among others) the following information:

- a description of the memory areas to be processed, including their size, hash size, comparison function, hashing function, conversion function, and printing procedure;
- a (possibly empty) ordered list of hiding and renaming operations to be applied sequentially; hiding and renaming operations can be intertwined arbitrarily;
- an (optional) hash-based cache table and (if applicable) statistical data about caching efficiency.

16.4 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_MASK_1;
```

This type denotes a pointer to the concrete representation of a masking object, which is supposed to be "opaque".

.....

```
void CAESAR_CREATE_MASK_1 (CAESAR_M, CAESAR_AREA, CAESAR_CACHE,
                           CAESAR_CACHE_SIZE, CAESAR_PRIME, CAESAR_COMPARE,
                           CAESAR_HASH, CAESAR_CONVERT, CAESAR_HISTORY)
   CAESAR_TYPE_MASK_1 *CAESAR_M;
   CAESAR_TYPE_AREA_1 CAESAR_AREA;
   CAESAR_TYPE_BOOLEAN CAESAR_CACHE;
   CAESAR_TYPE_NATURAL CAESAR_CACHE_SIZE;
   CAESAR_TYPE_BOOLEAN CAESAR_PRIME;
   CAESAR_TYPE_COMPARE_FUNCTION CAESAR_COMPARE;
   CAESAR_TYPE_HASH_FUNCTION CAESAR_HASH;
   CAESAR_TYPE_CONVERT_FUNCTION CAESAR_CONVERT;
   CAESAR_TYPE_BOOLEAN CAESAR_HISTORY;
   { ... }
```

This procedure allocates a masking object using `CAESAR_CREATE()` and assigns its address to `*CAESAR_M`. If the allocation fails or if any error occurs in this procedure, the `NULL` value is assigned to `*CAESAR_M`.

Note: because `CAESAR_TYPE_MASK_1` is a pointer type, any variable `CAESAR_M` of type `CAESAR_TYPE_MASK_1` must be allocated before used, for instance using:

```
CAESAR_CREATE_MASK_1 (&CAESAR_M, ...);
```

The value of `CAESAR_AREA` specifies the kind of memory areas that can be processed by the masking object to be created. All the memory areas handled using a given masking object must be of the same kind and have the same size and alignment.

The (optional) hash-based cache table is enabled if and only if CAESAR_CACHE is equal to CAESAR_TRUE.

If the cache table is disabled, the four next parameters CAESAR_CACHE_SIZE, CAESAR_PRIME, CAESAR_COMPARE, and CAESAR_HASH are unused and must be equal to 0, CAESAR_FALSE, NULL, and NULL, respectively; otherwise, the effect of this procedure is undefined. In the sequel, we only consider the case where the cache table is enabled.

Note: If the value of CAESAR_AREA is equal to CAESAR_STRING_AREA_1 (), the character strings that will be stored in the masking object pointed to by *CAESAR_M may be of arbitrary length. If the cache table is enabled, the contents of these character strings should remain constant, in the sense that, after a character string is passed as argument to the function CAESAR_APPLY_MASK_1() defined below, its contents should no longer be modified (as the cache table stores only the string pointer, but not the string contents).

The value of CAESAR_CACHE_SIZE determines the maximal number of areas that can be stored simultaneously in the cache table, together with the corresponding characters strings obtained after applying conversion, hiding, and/or renaming operations. Whenever a new area is inserted in the cache table and an existing area is already present with the same hash value as the new area, the new area will replace the existing area. If the value of CAESAR_CACHE_SIZE is equal to zero, it is replaced with a default value greater than zero.

If the value of CAESAR_PRIME is equal to CAESAR_TRUE and if the value of CAESAR_CACHE_SIZE is not a prime number, this value will be replaced by the nearest smaller prime number (since some hash functions require prime modulus). Otherwise, the value of CAESAR_CACHE_SIZE will be kept unchanged.

The actual value of the formal parameter CAESAR_COMPARE will be stored and associated to the masking object pointed to by *CAESAR_M. It will be used as a comparison function when it is necessary to decide whether two memory areas are equal or not. If the actual value of CAESAR_COMPARE is NULL, it may be replaced by a non-NULL default value depending on the value of CAESAR_AREA and according to the rules specified for function CAESAR_USE_COMPARE_FUNCTION_AREA_1() of the "area_1" library. In both cases, all area comparisons will be done using the CAESAR_COMPARE_AREA_1() macro defined in the "area_1" library.

The actual value of the formal parameter CAESAR_HASH will be stored and associated to the masking object pointed to by *CAESAR_M. It will be used as a hashing function when it is necessary to compute a hash-value for a memory area. If the actual value of CAESAR_HASH is NULL, it may be replaced by a non-NULL default value depending on the value of CAESAR_AREA and according to the rules specified for function CAESAR_USE_HASH_FUNCTION_AREA_1() of the "area_1" library. In both cases, area hashing will be done using the CAESAR_HASH_AREA_1() macro defined in the "area_1" library.

The actual value of the formal parameter CAESAR_CONVERT will be stored and associated to the masking object pointed to by *CAESAR_M. It will be used as a conversion function when it is necessary to translate a memory area into a character string. If the actual value of CAESAR_CONVERT is NULL, it may be replaced by a non-NULL default value depending on the value of CAESAR_AREA and according to the rules specified for function CAESAR_USE_CONVERT_FUNCTION_AREA_1() of the "area_1" library. In both cases, all area conversions will be done using the CAESAR_CONVERT_AREA_1() macro defined in the "area_1" library.

The value of the formal parameter CAESAR_HISTORY must be equal to CAESAR_TRUE if the user wants to invoke the CAESAR_HISTORY_MASK_1() function defined below.

Note: the list of hiding and renaming operations associated to the masking object pointed to by *CAESAR_M is not specified at the time the masking object is created but later, using the CAESAR_USE_HIDE_MASK_1(), CAESAR_USE_RENAME_MASK_1(), and

CAESAR_PARSE_OPTION_MASK_1() primitives defined below.

. . . . . .

```
void CAESAR_DELETE_MASK_1 (CAESAR_M)
   CAESAR_TYPE_MASK_1 *CAESAR_M;
   { ... }
```

This procedure frees the memory space corresponding to the masking object pointed to by *CAESAR_M using CAESAR_DELETE(). Afterwards, the NULL value is assigned to *CAESAR_M.

. . . . . .

```
void CAESAR_USE_HIDE_MASK_1 (CAESAR_M, CAESAR_PATHNAME, CAESAR_KIND,
                             CAESAR_FAILED)
   CAESAR_TYPE_MASK_1 CAESAR_M;
   CAESAR_TYPE_STRING CAESAR_PATHNAME;
   CAESAR_TYPE_NATURAL CAESAR_KIND;
   CAESAR_TYPE_BOOLEAN *CAESAR_FAILED;
   { ... }
```

This function adds a new hiding operation at the end of the (initially empty) list of hiding and renaming operations associated to the masking object pointed to by CAESAR_M.

The hiding operation is specified by the actual values of the formal parameters CAESAR_PATHNAME and CAESAR_KIND, the signification of which is the same as for function CAESAR_CREATE_HIDE_1() of the "hide_1" library.

Note: the additional formal parameters CAESAR_POSITIVE_HEADER and CAESAR_NEGATIVE_HEADER of CAESAR_CREATE_HIDE_1() are not accessible using CAESAR_USE_HIDE_MASK_1(); their default values will be used, meaning that the hiding file referred to by CAESAR_PATHNAME must start with either "`hide`" (positive header) or "`hide all but`" (negative header).

If the call to CAESAR_USE_HIDE_MASK_1() fails, (e.g., because the hiding file is incorrect), the boolean pointed to by *CAESAR_FAILED is set to CAESAR_TRUE, and an error message is printed to the standard output. Otherwise, the boolean pointed to by *CAESAR_FAILED it is set to CAESAR_FALSE.

If the cache table is enabled, its contents are erased after any call to CAESAR_USE_HIDE_MASK_1(), since the results of previous hiding and/or renaming operations performed so far can no longer be reused after a new hiding operation is added. This allows to invoke CAESAR_USE_HIDE_MASK_1() safely, even after a call to the function CAESAR_APPLY_MASK_1() defined below.

. . . . . .

```
void CAESAR_USE_RENAME_MASK_1 (CAESAR_M, CAESAR_PATHNAME, CAESAR_KIND,
                               CAESAR_FAILED)
    CAESAR_TYPE_MASK_1 CAESAR_M;
    CAESAR_TYPE_STRING CAESAR_PATHNAME;
    CAESAR_TYPE_NATURAL CAESAR_KIND;
    CAESAR_TYPE_BOOLEAN *CAESAR_FAILED;
```

```
{ ... }
```

This function adds a new renaming operation at the end of the (initially empty) list of hiding and renaming operations associated to the masking object pointed to by CAESAR_M.

The renaming operation is specified by the actual values of the formal parameters CAESAR_PATHNAME and CAESAR_KIND, the signification of which is the same as for function CAESAR_CREATE_RENAME_1() of the "rename_1" library.

Note: the additional formal parameter CAESAR_HEADER of CAESAR_CREATE_RENAME_1() is not accessible using CAESAR_USE_RENAME_MASK_1(); its default value will be used, meaning that the renaming file referred to by CAESAR_PATHNAME must start with "rename".

If the call to CAESAR_USE_RENAME_MASK_1() fails, (e.g., because the renaming file is incorrect), the boolean pointed to by *CAESAR_FAILED is set to CAESAR_TRUE, and an error message is printed to the standard output. Otherwise, the boolean pointed to by *CAESAR_FAILED it is set to CAESAR_FALSE.

If the cache table is enabled, its contents are erased after any call to CAESAR_USE_RENAME_MASK_1(), since the results of previous hiding and/or renaming operations performed so far can no longer be reused after a new renaming operation is added. This allows to invoke CAESAR_USE_RENAME_MASK_1() safely, even after a call to the function CAESAR_APPLY_MASK_1() defined below.

.....

```
void CAESAR_PARSE_OPTION_MASK_1 (CAESAR_M, CAESAR_ARGC, CAESAR_ARGV,
                                 CAESAR_NB_TOKENS)
   CAESAR_TYPE_MASK_1 CAESAR_M;
   CAESAR_TYPE_ARGC CAESAR_ARGC;
   CAESAR_TYPE_ARGV CAESAR_ARGV;
   CAESAR_TYPE_GENUINE_INT *CAESAR_NB_TOKENS;
   { ... }
```

This procedure parses the command-line options specified by parameters CAESAR_ARGC and CAESAR_ARGV and, if specific options defining a hiding or renaming operation are recognized, the corresponding hiding or renaming operation is added to the masking object pointed to by CAESAR_M; this addition is done by invoking the CAESAR_USE_HIDE_MASK_1() or CAESAR_USE_RENAME_MASK_1() procedure internally.

The parameters CAESAR_ARGC and CAESAR_ARGV specify a list of character strings (named options). They follow the standard conventions regarding the argc and argv parameters of the main() function of any C program. CAESAR_ARGC denotes the number of options and CAESAR_ARGV is a character string array containing the options.

A hiding operation is defined by the following sequence of command-line options:

`-hide [ -total | -partial | -gate ]`*hiding_filename*

where *hiding_filename* is the pathname of a hiding file (the format of this file is described in the "hide_1" library), and where the optional -total, -partial, and -gate options correspond, respectively, to the "total matching", "partial matching", and "gate matching" semantics defined in the documentation of the "hide_1" library; option -total is the default.

A renaming operation is defined by the following sequence of command-line options:

`-rename [ -total | -single | -multiple | -gate ]`*renaming_filename*

where *renaming_filename* is the pathname of a renaming file (the format of this file is described in the "rename_1" library), and where the optional `-total`, `-single`, `-multiple`, and `-gate` options correspond, respectively, to the "total matching", "single partial matching", "multiple partial matching", and "gate matching" semantics defined in the documentation of the "rename_1" library; option `-total` is the default.

The `CAESAR_PARSE_OPTION_MASK_1()` procedure processes command-line options from left to right. One single call to `CAESAR_PARSE_OPTION_MASK_1()` recognizes at most one (hiding or renaming) operation. Thus, `CAESAR_PARSE_OPTION_MASK_1()` should be used within a program loop in which `CAESAR_ARGC` is decremented and `CAESAR_ARGV` is incremented as hiding or renaming options are recognized.

An integer value is assigned to `*CAESAR_NB_TOKENS` in order to indicate how many command-line options have been recognized and processed. The conventions are the following:

- If `*CAESAR_NB_TOKENS` is a strictly positive integer N, the N-th first options have been recognized as forming either a hiding operation or a renaming operation. In this case, `CAESAR_ARGC` will need to be decremented by N and `CAESAR_ARGV` will need to be incremented by N.
- If `*CAESAR_NB_TOKENS` is equal to 0, then the first option of `CAESAR_ARGV` has not been recognized as forming either a hiding operation or a renaming operation.
- If `*CAESAR_NB_TOKENS` is strictly negative, the first options have been recognized as forming (at least, the beginning of) either a hiding operation or renaming operation, but some error occured while processing these options (e.g., incorrect option syntax, unreadable hiding or renaming file, memory shortage, ...). In this case, an error message will be printed to the standard output. This situation is usually considered as a fatal error.

.....

```
CAESAR_TYPE_STRING CAESAR_APPLY_MASK_1 (CAESAR_M, CAESAR_P)
   CAESAR_TYPE_MASK_1 CAESAR_M;
   CAESAR_TYPE_POINTER CAESAR_P;
   { ... }
```

This function first converts the memory area pointed to by `CAESAR_P` into a character string S using the conversion function associated to the masking object pointed to by `CAESAR_M`.

Then, this function applies, in sequence, to S the list of hiding and/or renaming operations associated to the masking object pointed to by `CAESAR_M` and returns the resulting character string.

Note: hiding is done using the `CAESAR_TEST_HIDE_1()` function of the "hide_1" library, a hidden character string being renamed into the constant character string `"i"`.

Note: renaming is done using the `CAESAR_TEST_RENAME_1()` operation of the "rename_1" library.

The order in which hiding and/or renaming operations are applied is the same as the order in which the procedures `CAESAR_USE_HIDE_MASK_1()` and/or `CAESAR_USE_RENAME_MASK_1()` have been called, successively, for the masking object pointed to by `CAESAR_M`.

If the list of operations is empty (i.e., if no calls to `CAESAR_USE_HIDE_1()` nor to `CAESAR_USE_RENAME_1()` have occurred), then the result is simply S.

Note: if enabled, the cache table will be used to speed up the execution of `CAESAR_APPLY_MASK_1()`.

Note: if the cache table is enabled and if the masking object was created to handle string areas, meaning that the `CAESAR_AREA` parameter of `CAESAR_CREATE_MASK_1()` was equal to `CAESAR_STRING_AREA_1()`, then `*CAESAR_P` is a pointer to a null-terminated character string. The contents of this string must remain constant, i.e., it is forbidden to modify them after invoking of `CAESAR_APPLY_MASK_1` until the masking object is deleted.

Note: as regards side effects, the contents of `CAESAR_M` and `CAESAR_P` are not modified, except for the cache table associated to `CAESAR_M`, if any, which might be modified by a call to to `CAESAR_APPLY_MASK_1()`.

Note: it is not allowed to modify the character string returned by `CAESAR_APPLY_MASK_1()` nor to free it, for instance using `free(3)`.

Note: the contents of the character string returned by `CAESAR_APPLY_MASK_1()` may be destroyed by a subsequent call to this function.

.....

```
#define CAESAR_LABEL_HIDE_FORMAT_MASK_1 "   with labels hidden using \"%s\""
```

This macro-definition returns the default value used for the parameter `CAESAR_HIDE_FORMAT` of function `CAESAR_HISTORY_MASK_1()` when the actual value given to this parameter is `NULL`.

.....

```
#define CAESAR_LABEL_RENAME_FORMAT_MASK_1 "   with labels renamed using \"%s\""
```

This macro-definition returns the default value used for the parameter `CAESAR_RENAME_FORMAT` of function `CAESAR_HISTORY_MASK_1()` when the actual value given to this parameter is `NULL`.

.....

```
CAESAR_TYPE_STRING CAESAR_HISTORY_MASK_1 (CAESAR_M, CAESAR_PREFIX,
                                          CAESAR_HIDE_FORMAT,
                                          CAESAR_RENAME_FORMAT,
                                          CAESAR_SUFFIX, CAESAR_SEPARATOR)
   CAESAR_TYPE_MASK_1 CAESAR_M;
   CAESAR_TYPE_STRING CAESAR_PREFIX;
   CAESAR_TYPE_STRING CAESAR_HIDE_FORMAT;
   CAESAR_TYPE_STRING CAESAR_RENAME_FORMAT;
   CAESAR_TYPE_STRING CAESAR_SUFFIX;
   CAESAR_TYPE_STRING CAESAR_SEPARATOR;
   { ... }
```

If the masking object pointed to by `CAESAR_M` was created by a call to `CAESAR_CREATE_MASK_1()` in which the value of the `CAESAR_HISTORY` parameter was equal to `CAESAR_FALSE`, the effect of this function is undefined.

Otherwise, this function allocates and returns a character string S summarizing the list of hiding and/or renaming operations associated to the masking object pointed to by `CAESAR_M`. If allocation

fails, then S will be equal to `NULL`; if not, S will consist of several components:

- If the character string pointed to by `CAESAR_PREFIX` is different from `NULL`, it will form the first component of S.
- Then, for each hiding or renaming operation associated to the masking object, a corresponding component will be appended to S; these operations are considered in the order in which the `CAESAR_USE_HIDE_MASK_1()` and `CAESAR_USE_RENAME_MASK_1()` procedures have been called.

  For a hiding operation, the corresponding component is given by the character string pointed to by `CAESAR_HIDE_FORMAT`. This character string follows the same conventions as a format string given to the `printf(3)` function. However, this string may contain at most one `%s` pattern, which will be substituted by the pathname of the hiding file associated to the operation. It should not contain any other pattern such as `%d`, `%x`, or even `%32s`. If `CAESAR_HIDE_FORMAT` is equal to `NULL`, it will be replaced by its default value `CAESAR_LABEL_HIDE_FORMAT_MASK_1`.

  For a renaming operation, the corresponding component is given by the character string pointed to by `CAESAR_RENAME_FORMAT`. This character string follows the same conventions as a format string given to the `printf(3)` function. However, this string may contain at most one `%s` pattern, which will be substituted by the pathname of the renaming file associated to the operation. It should not contain any other pattern such as `%d`, `%x`, or even `%32s`. If `CAESAR_RENAME_FORMAT` is equal to `NULL`, it will be replaced by its default value `CAESAR_LABEL_RENAME_FORMAT_MASK_1`.
- If the character string pointed to by `CAESAR_SUFFIX` is different from `NULL`, it will form the last component of S.
- The character string pointed to by `CAESAR_SEPARATOR` (or, if `CAESAR_SEPARATOR` is `NULL`, the default string `"\n"`) will be inserted as a separator between every two components of S.

Note: The resulting character string S is allocated using `CAESAR_CREATE()` and should be freed as soon as possible using `CAESAR_DELETE()`.

.....

```
void CAESAR_FAILURE_MASK_1 (CAESAR_M)
   CAESAR_TYPE_MASK_1 CAESAR_M;
   { ... }
```

This function returns the value of the failure counter (i.e., the number of searches that failed) of the cache table associated to the masking object pointed to by `CAESAR_M`.

If the cache table is not enabled in the masking object, the effect of this function is undefined.

.....

```
void CAESAR_SUCCESS_MASK_1 (CAESAR_M)
   CAESAR_TYPE_MASK_1 CAESAR_M;
   { ... }
```

This function returns the value of the success counter (i.e., the number of searches that succeeded) of the cache table associated to the masking object pointed to by `CAESAR_M`.

If the cache table is not enabled in the masking object, the effect of this function is undefined.

```
CAESAR_TYPE_FORMAT CAESAR_FORMAT_MASK_1 (CAESAR_M, CAESAR_FORMAT)
   CAESAR_TYPE_MASK_1 CAESAR_M;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the masking object pointed to by `CAESAR_M` will be printed by the procedure `CAESAR_PRINT_MASK_1()` (see below). Currently, the following formats are available:

- With format 0, statistical information about the masking object is displayed such as: the area size (in bytes), the area hashable size (in bytes), the area comparison function, the area hash function, the area conversion function, etc. The cache size is also displayed together with, if it is different from zero, the number of searches that failed and succeeded.
- With format 1, the list of operations performed by the masking object is displayed using the procedures `CAESAR_PRINT_HIDE_1()` and `CAESAR_PRINT_RENAME_1()` exported by the "hide_1" and "rename_1" libraries (these procedures are called with their respective formats set to 0).
- With format 2, the contents of the cache table are displayed if the cache table is enabled.
- (no other format available yet).

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

When called with `CAESAR_FORMAT` between 0 and 2, this fonction sets the current format of `CAESAR_M` 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_M` 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_M`. If `CAESAR_FORMAT` is equal to the constant `CAESAR_MAXIMAL_FORMAT`, the result is the maximal format value (i.e., 2). In all other cases, the effect of this function is undefined.

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_MASK_1 ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as it might be removed from future versions of the Open/Cæsar. Use function `CAESAR_FORMAT_MASK_1()` instead, called with argument `CAESAR_MAXIMAL_FORMAT`.

This function returns the maximal format value available for printing masking objects.

```
void CAESAR_PRINT_MASK_1 (CAESAR_FILE, CAESAR_M)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_MASK_1 CAESAR_M;
```

```
{ ... }
```

This procedure prints to file `CAESAR_FILE` a text containing information about the masking object pointed to by `CAESAR_M`. The nature of the information is determined by the current format of the masking object pointed to by `CAESAR_M`.

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

.....

Chapter 17

The "solve_1" library (version 1.3)

by Radu Mateescu

17.1 Purpose

The "solve_1" library provides primitives for solving alternation-free boolean equation systems, which are either provided "on the fly" or given as a text file encoded in the "BES" format. This library can be used as a back-end for various on the fly verification tools that formulate their corresponding problems (e.g., equivalence checking, model checking, tau-confluence reduction, etc.) in terms of boolean equation systems.

17.2 Usage

The "solve_1" library consists of:

- a predefined header file "caesar_solve_1.h";
- the precompiled library file "libcaesar.a", which implements the features described in "caesar_solve_1.h".

Note: The "solve_1" library is a software layer built above the primitives offered by the "standard", "area_1", "table_1", and "hash" libraries, and by the OPEN/CÆSAR graph module.

17.3 Boolean equation systems

See the "bes" manual page of CADP for:

- a definition of boolean equation systems and their terminology: equation blocks, boolean variables, conjunctive and disjunctive formulas, conjunctive and disjunctive variables, dependencies between variables, dependencies between blocks, alternation-free systems, block indexes, variable indexes, sign of a block, etc.
- a specification of the "BES" format that is used to store boolean equation systems in text files having the extension ".bes".

17.4 On the fly resolution

Given a boolean equation system, the on the fly (or local) resolution problem consists in computing the value of a particular variable defined in some block of the system. Contrary to global resolution, which consists in computing the values of all variables defined in the system (and therefore must examine all equations of the system), on the fly resolution computes the value of a variable without necessarily examining all equations of the system. This resolution technique allows to construct the boolean equation system in a demand-driven manner, and hence it is useful for building tools for on the fly verification, which explore one or more labelled transition systems incrementally.

The on the fly resolution method for alternation-free boolean equation systems implemented in the "solve_1" library proceeds as follows. To each equation block of the system is associated a resolution routine responsible for computing the values of the variables defined in the block. When the routine associated to a block is called to compute the value of a variable defined in that block, it may in turn call the routines associated to other blocks to compute the values of other variables defined in those blocks. Assuming that all resolution routines associated to the blocks will eventually terminate, the overall resolution process will also terminate, because the size of the call stack of resolution routines is bounded by the number N of blocks in the boolean equation system (since the system is alternation-free, there are no cyclic dependencies between blocks).

17.5 Boolean graphs

The resolution routines associated to the equation blocks are easier to develop using a representation of blocks as boolean graphs, which provide a more intuitive view of the dependencies between boolean variables. Given an equation block, its corresponding boolean graph is defined as follows:

- For each boolean variable occurring in the block, there is a vertex in the boolean graph.
- For each dependency from a boolean variable `Xi` to a boolean variable `Xj`, there is an edge "(`Xi`, `Xj`)" in the boolean graph.
- Each vertex of the boolean graph is labeled as disjunctive or conjunctive according to the kind of the boolean variable it denotes.

Boolean variables whose defining equations have right-hand side formulas identical to `false` (resp. `true`) are represented in the boolean graph as disjunctive (resp. conjunctive) vertices without successors. Dependencies between different equation blocks are represented as edges between the boolean graphs associated to the blocks.

The boolean graph associated to the boolean equation system above is shown below (edges are represented as couples of boolean variables, the kind of which is indicated between brackets).

```
(* graph for B0 *)
   (X0_0 [and], X1_0 [or])
   (X0_0 [and], X2_0 [and])
   (X1_0 [or],  X0_0 [and])
   (X1_0 [or],  X1_0 [or])
   (X1_0 [or],  X2_0 [and])
   (X2_0 [and], X0_1 [or])
   (X2_0 [and], X3_0 [or])
   (X3_0 [or],  X1_0 [or])
```

```
      (X3_0 [or],  X4_0 [and])

   (* graph for B1 *)
      (X0_1 [or],  X1_1 [or])
      (X0_1 [or],  X2_1 [and])
      (X2_1 [and], X2_1 [and])
      (X2_1 [and], X3_1 [or])
      (X3_1 [or],  X0_1 [or])
      (X3_1 [or],  X1_1 [or])
      (X3_1 [or],  X3_1 [or])
```

For each equation block, its corresponding resolution routine will explore forward the boolean graph associated to the block and will propagate backward the values of boolean variables already stabilized (i.e., the value of which has been determined). When solving a boolean variable, only the part of the boolean graph relevant for deciding the value of the variable is explored. For instance, the part of the boolean graph explored for solving variable `X0_0` of the boolean equation system above is shown below (all boolean variables contained in this part of the boolean graph are false).

```
   (* resolution graph for B0 *)
      (X0_0 [and], X1_0 [or])
      (X0_0 [and], X2_0 [and])
      (X1_0 [or],  X0_0 [and])
      (X1_0 [or],  X1_0 [or])
      (X1_0 [or],  X2_0 [and])
      (X2_0 [and], X0_1 [or])

   (* resolution graph for B1 *)
      (X0_1 [or],  X1_1 [or])
      (X0_1 [or],  X2_1 [and])
      (X2_1 [and], X2_1 [and])
```

The resolution was carried out as follows. After exploring variables `X0_0` and `X1_0`, the current unexplored successor of `X0_0` is `X2_0`. The exploration of `X2_0` is started by visiting variable `X0_1`, which is defined in block `B1`. This variable depends on `X1_1`, whose value is false (disjunctive vertex without successors) and `X2_1`, whose value is also false (conjunctive vertex with a self-loop in a minimal fixed point block). Thus, variable `X0_1` is false, and by propagating its value backward, variables `X2_0` and `X0_0` become false (conjunctive vertices with a successor equal to false).

17.6 Resolution modes

The "solve_1" library implements various algorithms (named "resolution modes") to solve boolean equation systems by exploring their associated boolean graphs. For efficiency reasons, each equation block of the same boolean equation system may be solved using the most appropriate algorithm. Currently, the following resolution modes are available:

- Mode 0 corresponds to a resolution algorithm based upon a depth-first search of the boolean graph associated to the equation block. This algorithm can be applied to any kind of equation block.
- Mode 1 corresponds to a resolution algorithm based upon a breadth-first search of the boolean graph associated to the equation block. This algorithm can be applied to any kind of equation

block. In practice, it performs slightly slower than mode 0, but produces diagnostics of smaller depth.

- Mode 2 corresponds to a resolution algorithm based upon a depth-first search of the boolean graph associated to the equation block. This algorithm can be applied only to acyclic equation blocks, i.e., without cyclic dependencies between variables. In practice, it consumes less memory than modes 0 and 1.

- Mode 3 corresponds to a resolution algorithm based upon a depth-first search of the boolean graph associated to the equation block. This algorithm can be applied only to disjunctive equation blocks, i.e., whose right-hand sides of equations are either (a) disjunctive formulas, or (b) conjunctive formulas whose operands are constants or variables defined in other blocks, with the possible exception of the last operand, which may be a variable defined in the current block (e.g., "`true and X1_j and false and X2`", where `X1_j` is defined in another block of index `j` and `X2` is defined in the current block). In practice, it consumes less memory than modes 0 and 1.

- Mode 4 corresponds to a resolution algorithm based upon a depth-first search of the boolean graph associated to the equation block. This algorithm can be applied only to conjunctive equation blocks, i.e., whose right-hand sides of equations are either (a) conjunctive formulas, or (b) disjunctive formulas whose operands are constants or variables defined in other blocks, with the possible exception of the last operand, which may be a variable defined in the current block (e.g., "`false or X1_j or true or X2`", where `X1_j` is defined in another block of index `j` and `X2` is defined in the current block). In practice, it consumes less memory than modes 0 and 1.

- Mode 5 corresponds to a resolution algorithm based upon a depth-first search of the boolean graph associated to the equation block. This algorithm can be applied to any kind of equation block. In practice, it exhibits a better performance than mode 0 when applied to equation blocks containing alternating dependencies between disjunctive and conjunctive variables (e.g., the equation blocks encoding equivalence checking problems).

- Mode 6 corresponds to a resolution algorithm based upon a breadth-first search of the boolean graph associated to the equation block. This algorithm can be applied only to disjunctive minimal fixed point equation blocks for which a single resolution was specified. In practice, it consumes less memory than mode 1 and produces diagnostics of smaller depth than mode 3.

- Mode 7 corresponds to a resolution algorithm based upon a breadth-first search of the boolean graph associated to the equation block. This algorithm can be applied only to conjunctive maximal fixed point equation blocks for which a single resolution was specified. In practice, it consumes less memory than mode 1 and produces diagnostics of smaller depth than mode 4.

- Mode 8 corresponds to a resolution algorithm based upon a depth-first search of the boolean graph associated to the equation block. This algorithm can be applied to any kind of equation block. In practice, it consumes less memory than modes 0 and 5, being especially useful for solving minimal (resp. maximal) fixed point equation blocks containing many conjunctive (resp. disjunctive) variables.

- Mode 9 corresponds to a resolution algorithm based upon a breadth-first search of the boolean graph associated to the equation block. This algorithm can be applied only to acyclic equation blocks, i.e., without cyclic dependencies between variables. When applied to an acyclic block consisting of singular equations (i.e., having only one boolean variable in their right-hand sides),

whose boolean graph is a sequence, this algorithm consumes a bounded amount of memory, independent of the number of equations in the block (length of the sequence).

- (no other resolution mode available yet)

17.7 Internal representation

To enable on the fly exploration, the boolean graphs associated to the equation blocks of a boolean equation system are represented in a generic, implicit manner using a scheme similar to the one defined by the OPEN/CÆSAR graph module for representing labelled transition systems. This representation roughly consists of the following ingredients (see procedure `CAESAR_CREATE_SOLVE_1()` below for additional details):

- Boolean variables (vertices of the boolean graph) are represented as pointers to memory areas of fixed size (for each equation block, all variables defined in that block must have the same size). The precise meaning of the variable contents is defined by the application program and is not relevant for the resolution algorithms.
- Each equation block is equipped with several functions computing various information about the variables defined in the block: a function returning the kind of a variable (disjunctive or conjunctive), a comparison function, a hashing function, a printing function, and an iterator procedure which enumerates the successors of a variable in the boolean graph.
- Application programs may also associate, to each edge of the boolean graph, a specific information (label) represented as a pointer to a memory area. The contents of these memory areas are not meaningful for the resolution algorithms, which manipulate the pointers to these areas only by copying them and (possibly) by comparing them to `NULL`.

To speed up the overall resolution process, each equation block of the boolean equation system has associated an internal table which stores the boolean variables already explored during the previous calls of the resolution routine associated to the block. This avoids recomputations of boolean variables by subsequent calls of the same resolution routine, leading to an overall resolution process of time complexity linear in the size of the boolean equation system (number of variables and operators).

17.8 Diagnostic generation

A diagnostic for a boolean variable is a boolean subgraph rooted at the vertex corresponding to the variable, which illustrates the truth value computed for the variable. If the boolean variable is `true` (resp. `false`), then the diagnostic is called example (resp. counterexample). Disjunctive (resp. conjunctive) vertices belonging to an example have only one successor (resp. all their successors) contained in the example. Disjunctive (resp. conjunctive) vertices belonging to a counterexample have all their successors (resp. only one successor) contained in the counterexample.

The diagnostic of a boolean variable defined in an equation block is always contained in the part of the boolean graph explored by the resolution routine associated to that block when it solved the variable. A counterexample for variable `X0_0` of the boolean equation system above is shown below.

```
(* diagnostic graph for X0_0 in B0 *)
   (X0_0 [and], X2_0 [and])
```

```
      (X2_0 [and], X0_1 [or])

   (* diagnostic graph for X0_1 in B1 *)
      (X0_1 [or],  X1_1 [or])
      (X0_1 [or],  X2_1 [and])
      (X2_1 [and], X2_1 [and])
```

To speed up the generation of diagnostics, resolution routines also compute diagnostic-related information which is kept in the internal tables associated to the blocks. Diagnostics are represented in a generic, implicit manner using a scheme similar to the one defined by the OPEN/CÆSAR graph module for representing labelled transition systems. This representation is based upon an iterator procedure that enumerates the successors of a variable contained in its diagnostic (see procedure `CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1()` below).

Application programs can generate diagnostics by exploring them using this iterator procedure. To spare memory, the successors of a boolean variable contained in its diagnostic are provided by the iterator procedure as pointers to variables that were stored in the internal tables when the variable was solved (see procedure `CAESAR_START_DIAGNOSTIC_SOLVE_1()` below). Consequently, these variables should neither be modified, nor freed by the application program.

Note: Edges contained in diagnostics preserve the information (label) that was attached to the edges of the boolean graph by the application program (see procedure `CAESAR_CREATE_SOLVE_1()` below).

17.9 Description

The "solve_1" library allows to create and handle boolean equation systems on the fly, providing procedures for resolution, inspection, diagnostic generation, reading from, and writing to text files.

17.10 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_SOLVE_1;
```

This type denotes a pointer to the concrete representation of a boolean equation system. This representation is supposed to be "opaque".

.....

```
typedef CAESAR_TYPE_BOOLEAN CAESAR_TYPE_BLOCK_SIGN_SOLVE_1;
```

This type indicates the sign (minimal or maximal fixed point) associated to an equation block of a boolean equation system.

.....

```
#define CAESAR_MINIMAL_FIXED_POINT_SOLVE_1 \
   ((CAESAR_TYPE_BLOCK_SIGN_SOLVE_1) CAESAR_TRUE)
```

This constant denotes the minimal fixed point sign.

```
#define CAESAR_MAXIMAL_FIXED_POINT_SOLVE_1 \
   ((CAESAR_TYPE_BLOCK_SIGN_SOLVE_1) CAESAR_FALSE)
```

This constant denotes the maximal fixed point sign.

```
typedef CAESAR_TYPE_BOOLEAN CAESAR_TYPE_VARIABLE_KIND_SOLVE_1;
```

This type indicates the kind (disjunctive or conjunctive) associated to a boolean variable.

```
#define CAESAR_DISJUNCTIVE_VARIABLE_SOLVE_1 \
   ((CAESAR_TYPE_VARIABLE_KIND_SOLVE_1) CAESAR_TRUE)
```

This constant denotes the disjunctive variable kind.

```
#define CAESAR_CONJUNCTIVE_VARIABLE_SOLVE_1 \
   ((CAESAR_TYPE_VARIABLE_KIND_SOLVE_1) CAESAR_FALSE)
```

This constant denotes the conjunctive variable kind.

```
typedef CAESAR_TYPE_BLOCK_SIGN_SOLVE_1
   (*CAESAR_TYPE_BLOCK_SIGN_FUNCTION_SOLVE_1) (CAESAR_TYPE_NATURAL);
```

This type denotes a pointer to a function which takes as parameter a natural number (index of an equation block) and returns the sign (minimal or maximal fixed point) of the block.

```
typedef CAESAR_TYPE_VARIABLE_KIND_SOLVE_1
   (*CAESAR_TYPE_VARIABLE_KIND_FUNCTION_SOLVE_1) (CAESAR_TYPE_POINTER);
```

This type denotes a pointer to a function which takes as parameter a pointer to a boolean variable and returns the kind (disjunctive or conjunctive) of the variable.

```
typedef CAESAR_TYPE_AREA_1
   (*CAESAR_TYPE_AREA_FUNCTION_SOLVE_1) (CAESAR_TYPE_NATURAL);
```

This type denotes a pointer to a function which takes as parameter a natural number (index of an equation block) and returns the area (size and alignment) of the boolean variables defined in that block.

.....

```
typedef CAESAR_TYPE_BOOLEAN
   (*CAESAR_TYPE_BOOLEAN_FUNCTION_SOLVE_1) (CAESAR_TYPE_NATURAL);
```

This type denotes a pointer to a function which takes as parameter a natural number and returns a boolean value.

.....

```
typedef CAESAR_TYPE_NATURAL
   (*CAESAR_TYPE_NATURAL_FUNCTION_SOLVE_1) (CAESAR_TYPE_NATURAL);
```

This type denotes a pointer to a function which takes as parameter a natural number and returns a natural number.

.....

```
typedef enum {
     CAESAR_NONE_SOLVE_1,
     CAESAR_MULTIPLE_RESOLUTION_SOLVE_1,
     CAESAR_MEMORY_SHORTAGE_SOLVE_1,
     CAESAR_RECURSIVE_BLOCK_SOLVE_1,
     CAESAR_CYCLIC_BLOCK_SOLVE_1,
     CAESAR_NOT_DISJUNCTIVE_BLOCK_SOLVE_1,
     CAESAR_NOT_CONJUNCTIVE_BLOCK_SOLVE_1,
     CAESAR_MINIMAL_FIXED_POINT_BLOCK_SOLVE_1,
     CAESAR_MAXIMAL_FIXED_POINT_BLOCK_SOLVE_1
}    CAESAR_TYPE_ERROR_SOLVE_1;
```

This enumerated type defines the error codes produced as a side effect by calls to the function `CAESAR_COMPUTE_SOLVE_1()` (see below), which performs the resolution of a boolean variable defined in a block of a boolean equation system. The error codes have the following meaning:

- `CAESAR_NONE_SOLVE_1` indicates that the resolution was performed successfully.
- `CAESAR_MULTIPLE_RESOLUTION_SOLVE_1` indicates that another resolution of a variable of the block was already performed, whereas at the creation of the boolean equation system (see procedure `CAESAR_CREATE_SOLVE_1()` below) a single resolution was specified for the block. If one of the resolution algorithms 6 or 7, dedicated to blocks with single resolution, was specified for the block, this error code is produced regardless of the fact that a single resolution was specified or not for the block.

- CAESAR_MEMORY_SHORTAGE_SOLVE_1 indicates that a memory allocation failed during the resolution.
- CAESAR_RECURSIVE_BLOCK_SOLVE_1 indicates the presence of a cyclic dependency between the blocks of the boolean equation system, which violates the alternation-free condition.
- CAESAR_CYCLIC_BLOCK_SOLVE_1 indicates the presence of a cyclic dependency between the variables defined in the block, whereas at the creation of the boolean equation system (see procedure CAESAR_CREATE_SOLVE_1() below) the resolution algorithm 2, dedicated to acyclic blocks, was specified for the block.
- CAESAR_NOT_DISJUNCTIVE_BLOCK_SOLVE_1 indicates that the current block is not disjunctive, whereas at the creation of the boolean equation system (see procedure CAESAR_CREATE_SOLVE_1() below) one of the resolution algorithms 3 or 6, dedicated to disjunctive blocks, was specified for the block.
- CAESAR_NOT_CONJUNCTIVE_BLOCK_SOLVE_1 indicates that the current block is not conjunctive, whereas at the creation of the boolean equation system (see procedure CAESAR_CREATE_SOLVE_1() below) one of the resolution algorithms 4 or 7, dedicated to conjunctive blocks, was specified for the block.
- CAESAR_MINIMAL_FIXED_POINT_BLOCK_SOLVE_1 indicates that the current block denotes a minimal fixed point, whereas at the creation of the boolean equation system (see procedure CAESAR_CREATE_SOLVE_1() below) the resolution algorithm 7, dedicated to maximal fixed point blocks, was specified for the block.
- CAESAR_MAXIMAL_FIXED_POINT_BLOCK_SOLVE_1 indicates that the current block denotes a maximal fixed point, whereas at the creation of the boolean equation system (see procedure CAESAR_CREATE_SOLVE_1() below) the resolution algorithm 6, dedicated to minimal fixed point blocks, was specified for the block.

Note: The error code produced by a call to CAESAR_COMPUTE_SOLVE_1() can be obtained by using the function CAESAR_STATUS_COMPUTE_SOLVE_1() (see below).

.....

```
void CAESAR_CREATE_SOLVE_1 (CAESAR_B,
                            CAESAR_NUMBER_OF_BLOCKS,
                            CAESAR_BLOCK_SIGN,
                            CAESAR_BLOCK_UNIQUE_RESOLUTION,
                            CAESAR_BLOCK_SOLVE_MODE,
                            CAESAR_BLOCK_VARIABLE_AREA,
                            CAESAR_BLOCK_LIMIT_SIZE,
                            CAESAR_BLOCK_HASH_SIZE,
                            CAESAR_BLOCK_PRIME,
                            CAESAR_VARIABLE_KIND,
                            CAESAR_VARIABLE_COMPARE,
                            CAESAR_VARIABLE_HASH,
                            CAESAR_VARIABLE_PRINT,
                            CAESAR_VARIABLE_ITERATE,
                            CAESAR_INFO)
```

```
CAESAR_TYPE_SOLVE_1 *CAESAR_B;
CAESAR_TYPE_NATURAL CAESAR_NUMBER_OF_BLOCKS;
CAESAR_TYPE_BLOCK_SIGN_FUNCTION_SOLVE_1 CAESAR_BLOCK_SIGN;
CAESAR_TYPE_BOOLEAN_FUNCTION_SOLVE_1 CAESAR_BLOCK_UNIQUE_RESOLUTION;
CAESAR_TYPE_NATURAL_FUNCTION_SOLVE_1 CAESAR_BLOCK_SOLVE_MODE;
CAESAR_TYPE_AREA_FUNCTION_SOLVE_1 CAESAR_BLOCK_VARIABLE_AREA;
CAESAR_TYPE_NATURAL_FUNCTION_SOLVE_1 CAESAR_BLOCK_LIMIT_SIZE;
CAESAR_TYPE_NATURAL_FUNCTION_SOLVE_1 CAESAR_BLOCK_HASH_SIZE;
CAESAR_TYPE_BOOLEAN_FUNCTION_SOLVE_1 CAESAR_BLOCK_PRIME;
CAESAR_TYPE_VARIABLE_KIND_FUNCTION_SOLVE_1 CAESAR_VARIABLE_KIND;
CAESAR_TYPE_COMPARE_FUNCTION CAESAR_VARIABLE_COMPARE;
CAESAR_TYPE_HASH_FUNCTION CAESAR_VARIABLE_HASH;
CAESAR_TYPE_PRINT_FUNCTION CAESAR_VARIABLE_PRINT;
void (*CAESAR_VARIABLE_ITERATE) (CAESAR_TYPE_POINTER, CAESAR_TYPE_POINTER,
   void (*) (CAESAR_TYPE_POINTER, CAESAR_TYPE_NATURAL, CAESAR_TYPE_POINTER));
CAESAR_TYPE_POINTER CAESAR_INFO;
{ ... }
```

This procedure allocates a boolean equation system using `CAESAR_CREATE()` and assigns its address to `*CAESAR_B`. If the allocation fails, the `NULL` value is assigned to `*CAESAR_B`.

Note: Because `CAESAR_TYPE_SOLVE_1` is a pointer type, any variable `CAESAR_B` of type `CAESAR_TYPE_SOLVE_1` must be allocated before used, for instance using:

```
CAESAR_CREATE_SOLVE_1 (&CAESAR_B, ...);
```

The value of `CAESAR_NUMBER_OF_BLOCKS` determines the number of equation blocks contained in the boolean equation system. Each equation block will be assigned an unique index in the range 0..`CAESAR_NUMBER_OF_BLOCKS` − 1. If the value of `CAESAR_NUMBER_OF_BLOCKS` is zero, the effect is undefined.

The actual value of the formal parameter `CAESAR_BLOCK_SIGN` will be stored and associated to the boolean equation system pointed to by `*CAESAR_B`. It will be used to assign to each equation block its corresponding sign, indicating whether the block denotes a minimal or a maximal fixed point.

Precisely, the actual value of `CAESAR_BLOCK_SIGN` should be a pointer to a function with a parameter `caesar_block` that returns `CAESAR_MINIMAL_FIXED_POINT_SOLVE_1` (resp. `CAESAR_MAXIMAL_FIXED_POINT_SOLVE_1`) if the equation block of index `caesar_block` denotes a minimal fixed point (resp. a maximal fixed point), where `caesar_block` is in the range 0..`CAESAR_NUMBER_OF_BLOCKS` − 1.

The actual value of the formal parameter `CAESAR_BLOCK_UNIQUE_RESOLUTION` will be stored and associated to the boolean equation system pointed to by `*CAESAR_B`. It will be used to assign to each equation block a boolean indicating whether only one variable (or several variables) of the block will be solved.

Precisely, the actual value of `CAESAR_BLOCK_UNIQUE_RESOLUTION` should be a pointer to a function `caesar_f` with a parameter `caesar_block` that returns `CAESAR_TRUE` (resp. `CAESAR_FALSE`) if only one variable (resp. several variables) of the equation block of index `caesar_block` will be solved, where `caesar_block` is in the range 0..`CAESAR_NUMBER_OF_BLOCKS` − 1.

Note: If only one variable of a block will be solved, the function `caesar_f` may return for that block either `CAESAR_TRUE`, or `CAESAR_FALSE`, without influencing the resolution result; however, returning `CAESAR_TRUE` may increase the performance of some resolution algorithms.

Note: The equation blocks encoding equivalence checking problems are typical examples of blocks for which only one variable must be solved, namely the variable representing the equivalence between the initial states of two labelled transition systems.

The actual value of the formal parameter `CAESAR_BLOCK_SOLVE_MODE` will be stored and associated to the boolean equation system pointed to by `*CAESAR_B`. It will be used to assign to each equation block its corresponding resolution mode, determining which algorithm will be used by the resolution routine associated to the block.

Precisely, the actual value of `CAESAR_BLOCK_SOLVE_MODE` should be a pointer to a function `caesar_f` with a parameter `caesar_block` that returns the resolution mode associated to the equation block of index `caesar_block`, where `caesar_block` is in the range 0..`CAESAR_NUMBER_OF_BLOCKS` − 1.

If the resolution mode returned by `caesar_f` for some value of `caesar_block` is not among the aforementioned list of available resolution modes, the effect is undefined.

Note: The resolution algorithms denoted by modes 0, 1, 2, 3, and 4 are described in the publications [Mat03,Mat06a], where they are named A1, A2, A3, and A4. The correspondence between modes and the names of the algorithms is the following: mode 0 corresponds to A1; mode 1 corresponds to A2; mode 2 corresponds to A3; modes 3 and 4, which are symmetric, correspond to A4.

Note: If the boolean equation system pointed to by `*CAESAR_B` contains several equation blocks, each block may have associated a different resolution mode. In practice, this is useful for independently optimizing the resolution of blocks having a particular structure (e.g., acyclic, disjunctive, conjunctive).

The actual value of the formal parameter `CAESAR_BLOCK_VARIABLE_AREA` will be stored and associated to the boolean equation system pointed to by `*CAESAR_B`. It will be used to assign to each equation block the (constant) size and (constant) alignment factor of the boolean variables defined in the block.

Precisely, the actual value of `CAESAR_BLOCK_VARIABLE_AREA` should be a pointer to a function with a parameter `caesar_block` that returns the area (which indicates the length and alignment factor) of the boolean variables defined in the equation block of index `caesar_block`, where `caesar_block` is in the range 0..`CAESAR_NUMBER_OF_BLOCKS` − 1.

The actual value of the formal parameter `CAESAR_BLOCK_LIMIT_SIZE` will be stored and associated to the boolean equation system pointed to by `*CAESAR_B`. It will be used to assign to each equation block the maximal size (number of items) of the internal table associated to the block.

Precisely, the actual value of `CAESAR_BLOCK_LIMIT_SIZE` should be a pointer to a function `caesar_f` with a parameter `caesar_block` that returns the maximal size (number of items) of the internal table associated to the equation block of index `caesar_block`, where `caesar_block` is in the range 0..`CAESAR_NUMBER_OF_BLOCKS` − 1. The value returned by `caesar_f` must be less or equal to a predefined value M (see the "table_1" library). If it is equal to zero, it is replaced by the default value M.

The actual value of the formal parameter `CAESAR_BLOCK_HASH_SIZE` will be stored and associated to the boolean equation system pointed to by `*CAESAR_B`. It will be used to assign to each equation block the size (number of entries) of the hash-table accompanying the internal table associated to the block.

Precisely, the actual value of `CAESAR_BLOCK_HASH_SIZE` should be a pointer to a function `caesar_f` with a parameter `caesar_block` that returns the size (number of entries) of the hash-table accompanying the internal table associated to the equation block of index `caesar_block`, where `caesar_block` is in the range 0..`CAESAR_NUMBER_OF_BLOCKS` − 1. If the value returned by `caesar_f` for some value of `caesar_block` is zero, it is replaced with a default value greater than zero (see the "table_1" library).

The actual value of the formal parameter CAESAR_BLOCK_PRIME will be stored and associated to the boolean equation system pointed to by *CAESAR_B. It will be used to assign to each equation block a boolean value allowing to adjust the size of the hash-table accompanying the internal table associated to the block.

Precisely, the actual value of CAESAR_BLOCK_PRIME should be a pointer to a function caesar_f with a parameter caesar_block that returns a boolean value which will be stored and associated to the equation block of index caesar_block, where caesar_block is in the range 0..CAESAR_NUMBER_OF_BLOCKS−1. If the value returned by caesar_f for some value of caesar_block is equal to CAESAR_TRUE and if the value returned by CAESAR_BLOCK_HASH_SIZE for that block is not a prime number, this value will be replaced by the nearest smaller prime number (since some hash functions require prime modulus). Otherwise, the value returned by CAESAR_BLOCK_HASH_SIZE for that block will be kept unchanged (see the "table_1" library).

The actual value of the formal parameter CAESAR_VARIABLE_KIND will be stored and associated to the boolean equation system pointed to by *CAESAR_B. It will be used as a function returning the kind (disjunctive or conjunctive) of the boolean variables defined in an equation block of the system.

Precisely, the actual value of CAESAR_VARIABLE_KIND should be a pointer to a function caesar_f with a parameter caesar_variable that returns CAESAR_DISJUNCTIVE_VARIABLE_SOLVE_1 (resp. CAESAR_CONJUNCTIVE_VARIABLE_SOLVE_1) if the boolean variable pointed to by caesar_variable is disjunctive (resp. conjunctive). The index of the equation block in which the boolean variable pointed to by caesar_variable is defined can be obtained within caesar_f by calling the function CAESAR_CURRENT_BLOCK_SOLVE_1() (see below). A pointer to the boolean equation system containing this block (i.e., the value assigned to *CAESAR_B) can be obtained within caesar_f by calling the function CAESAR_CURRENT_SYSTEM_SOLVE_1() (see below).

The actual value of the formal parameter CAESAR_VARIABLE_COMPARE will be stored and associated to the boolean equation system pointed to by *CAESAR_B. It will be used as a comparison function for the boolean variables defined in an equation block of the system.

Precisely, the actual value of CAESAR_VARIABLE_COMPARE should be a pointer to a comparison function caesar_f with two parameters caesar_variable_1 and caesar_variable_2 that returns CAESAR_TRUE (resp. CAESAR_FALSE) if the boolean variables pointed to by caesar_variable_1 and caesar_variable_2 are equal (resp. different). The index of the equation block in which the boolean variables pointed to by caesar_variable_1 and caesar_variable_2 are defined can be obtained within caesar_f by calling the function CAESAR_CURRENT_BLOCK_SOLVE_1() (see below). A pointer to the boolean equation system containing this block (i.e., the value assigned to *CAESAR_B) can be obtained within caesar_f by calling the function CAESAR_CURRENT_SYSTEM_SOLVE_1() (see below).

The actual value of the formal parameter CAESAR_VARIABLE_HASH will be stored and associated to the boolean equation system pointed to by *CAESAR_B. It will be used as a hash-function for the boolean variables defined in an equation block of the system.

Precisely, the actual value of CAESAR_VARIABLE_HASH should be a pointer to a hash function caesar_f with two parameters caesar_variable and caesar_modulus that returns a hash-value computed on the byte string caesar_variable [0] up to caesar_variable [caesar_size - 1], where the actual value of caesar_size will always be equal to the size of the boolean variable pointed to by caesar_variable. This hash-value must belong to the range 0..caesar_modulus − 1. The index of the equation block in which the boolean variable pointed to by caesar_variable is defined can be obtained within caesar_f by calling the function CAESAR_CURRENT_BLOCK_SOLVE_1() (see below). A pointer to the boolean equation system containing this block (i.e., the value assigned to *CAESAR_B) can be obtained within caesar_f by calling the function CAESAR_CURRENT_SYSTEM_SOLVE_1() (see below).

The actual value of the formal parameter CAESAR_VARIABLE_PRINT will be stored and associated to the boolean equation system pointed to by *CAESAR_B. It will be used as a printing procedure for the boolean variables defined in an equation block of the system.

Precisely, the actual value of CAESAR_VARIABLE_PRINT should be a pointer to a printing procedure caesar_p with two parameters caesar_file and caesar_variable that prints to file caesar_file information about the contents of the boolean variable pointed to by caesar_variable. The index of the equation block in which the boolean variable pointed to by caesar_variable is defined can be obtained within caesar_p by calling the function CAESAR_CURRENT_BLOCK_SOLVE_1() (see below). A pointer to the boolean equation system containing this block (i.e., the value assigned to *CAESAR_B) can be obtained within caesar_p by calling the function CAESAR_CURRENT_SYSTEM_SOLVE_1() (see below).

The actual value of the formal parameter CAESAR_VARIABLE_ITERATE will be stored and associated to the boolean equation system pointed to by *CAESAR_B. It will be used as an iterator procedure enumerating all successors of the boolean variables defined in an equation block of the system.

Any user-defined procedure caesar_p can be used as an actual value for formal parameter CAESAR_VARIABLE_ITERATE, provided that its declaration has the form:

```
void caesar_p (caesar_variable_1, caesar_variable_2, caesar_loop)
   CAESAR_TYPE_POINTER caesar_variable_1;
   CAESAR_TYPE_POINTER caesar_variable_2;
   void (*caesar_loop) (CAESAR_TYPE_POINTER, CAESAR_TYPE_NATURAL,
      CAESAR_TYPE_POINTER);
   { ... }
```

This procedure caesar_p enumerates all successors of the boolean variable pointed to by caesar_variable_1. The index of the equation block in which the boolean variable pointed to by caesar_variable_1 is defined can be obtained within caesar_p by calling the function CAESAR_CURRENT_BLOCK_SOLVE_1() (see below). A pointer to the boolean equation system containing this block (i.e., the value assigned to *CAESAR_B) can be obtained within caesar_p by calling the function CAESAR_CURRENT_SYSTEM_SOLVE_1() (see below). At each iteration performed by caesar_p, two actions must be carried out:

- First, the boolean variable pointed to by caesar_variable_2 must be assigned a new value, such that "(caesar_variable_1, caesar_variable_2)" is an edge of the boolean graph.
- Second, the procedure pointed to by caesar_loop must be called. The actual value of the formal parameter caesar_loop is a procedure caesar_q whose declaration has the form:

  ```
  void caesar_q (caesar_label, caesar_block_2, caesar_variable_2)
     CAESAR_TYPE_POINTER caesar_label;
     CAESAR_TYPE_NATURAL caesar_block_2;
     CAESAR_TYPE_POINTER caesar_variable_2;
     { ... }
  ```

  Therefore, each call to the procedure pointed to by caesar_loop must have the following parameters:

  ```
  (*caesar_loop) (caesar_label, caesar_block_2, caesar_variable_2)
  ```

  Parameter caesar_label is either a pointer to a memory area containing additional information associated to the edge "(caesar_variable_1, caesar_variable_2)" of the boolean graph, or

is equal to NULL if no such information is desired. Parameter caesar_block_2 is the index of the equation block where the boolean variable caesar_variable_2 is defined.

Note: The memory area pointed to by the parameter caesar_variable_1 contains a boolean variable and should neither be modified, nor freed by the procedure caesar_p.

Note: The memory area pointed to by the parameter caesar_variable_2 is already allocated and should not be freed by the procedure caesar_p.

Note: The actual value passed to the parameter caesar_label when the procedure pointed to by caesar_loop is invoked by caesar_p is meaningless with respect to boolean resolution (the value passed to caesar_label will only be copied and possibly compared to NULL by the resolution algorithms). The parameter caesar_label allows to attach application-specific information to the edges going out of a boolean variable; this information is retrieved in the diagnostic generated for that variable. For instance, when using the "solve_1" library for model checking, caesar_label may contain a pointer to a label of the labelled transition system on which a temporal logic formula is verified. It is the users' responsibility to manage the memory area pointed to by caesar_label; in particular, it is recommended not to free this memory area until the resolution of the boolean equation system is finished.

The value of CAESAR_INFO has no effect on the execution of procedure CAESAR_CREATE_SOLVE_1(). Parameter CAESAR_INFO is intended to serve for future extensions of this procedure; when using the current version of the "solve_1" library, it is recommended to set this parameter to NULL.

. . . . . .

```
CAESAR_TYPE_SOLVE_1 CAESAR_CURRENT_SYSTEM_SOLVE_1 ()
   { ... }
```

This function returns a pointer to the boolean equation system which is currently under resolution. It should be called only within the functions and procedures given as actual values for the formal parameters CAESAR_VARIABLE_KIND, CAESAR_VARIABLE_COMPARE, CAESAR_VARIABLE_HASH, CAESAR_VARIABLE_PRINT, and CAESAR_VARIABLE_ITERATE of procedure CAESAR_CREATE_SOLVE_1() (see above); in this case, the result is a pointer to the boolean equation system created by the call to CAESAR_CREATE_SOLVE_1(). If this function is called anywhere else in the application program, the result is undefined.

Note: This function allows to invoke, within the five aforementioned functions and procedures, various primitives of the "solve_1" library on the current boolean equation system (e.g., resolution, printing, etc.).

. . . . . .

```
CAESAR_TYPE_NATURAL CAESAR_CURRENT_BLOCK_SOLVE_1 ()
   { ... }
```

This function returns the index of the equation block which is currently under resolution; this block is in turn contained in the boolean equation system which is currently under resolution, pointed to by the result of function CAESAR_CURRENT_SYSTEM_SOLVE_1() (see above). It should be called only within the functions and procedures given as actual values for the formal parameters CAESAR_VARIABLE_KIND, CAESAR_VARIABLE_COMPARE, CAESAR_VARIABLE_HASH,

CAESAR_VARIABLE_PRINT, and CAESAR_VARIABLE_ITERATE of procedure CAESAR_CREATE_SOLVE_1() (see above); in this case, the result is the index of the block, i.e., a natural number in the range $0..N-1$, where N is the number of blocks in the boolean equation system created by the call to CAESAR_CREATE_SOLVE_1(). If this function is called anywhere else in the application program, the result is undefined.

Note: This function allows to identify the block in which the boolean variable(s) passed as arguments to the five aforementioned functions and procedures are defined, and thus to handle these variables accordingly (the size and the contents of variables defined in different blocks may differ). It is especially useful when the number of blocks in the boolean equation system is unknown statically (e.g., when using the "solve_1" library for model checking, the number of blocks is inferred from a temporal logic formula read as input).

.....

```
void CAESAR_DELETE_SOLVE_1 (CAESAR_B)
   CAESAR_TYPE_SOLVE_1 *CAESAR_B;
   { ... }
```

This procedure frees the memory space corresponding to the boolean equation system pointed to by *CAESAR_B using CAESAR_DELETE(). The boolean variables stored in internal tables allocated during previous resolutions (if any) of the boolean equation system are also freed. Afterwards, the NULL value is assigned to *CAESAR_B.

.....

```
void CAESAR_PURGE_BLOCK_SOLVE_1 (CAESAR_B, CAESAR_I)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   { ... }
```

This procedure reinitializes the information associated to the equation block of index CAESAR_I of the boolean equation system pointed to by CAESAR_B. The internal table associated to the block is emptied using CAESAR_PURGE_TABLE_1(). Afterwards, the block is exactly in the same state as after the creation of the boolean equation system using CAESAR_CREATE_SOLVE_1().

If the block index CAESAR_I is outside the range $0..N-1$ (where N is the number of blocks in the system), the result is undefined.

.....

```
CAESAR_TYPE_BOOLEAN CAESAR_COMPUTE_SOLVE_1 (CAESAR_B, CAESAR_I, CAESAR_V)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   CAESAR_TYPE_POINTER CAESAR_V;
   { ... }
```

This function computes the value of the boolean variable pointed to by CAESAR_V, which must be defined in the equation block of index CAESAR_I of the boolean equation system pointed to by CAESAR_B.

It also sets a field of type `CAESAR_TYPE_ERROR_SOLVE_1` associated to the block, indicating whether the resolution was carried out successfully or not; this field can be inspected using the function `CAESAR_STATUS_COMPUTE_SOLVE_1()` (see below).

If the block index `CAESAR_I` is outside the range $0..N-1$ (where N is the number of blocks in the system), the result is undefined.

.....

```
CAESAR_TYPE_ERROR_SOLVE_1 CAESAR_STATUS_COMPUTE_SOLVE_1 (CAESAR_B, CAESAR_I)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   { ... }
```

This function returns the status of the last resolution performed by a call to the function `CAESAR_COMPUTE_SOLVE_1()` (see above) on a boolean variable defined in the equation block of index `CAESAR_I` of the boolean equation system pointed to by `CAESAR_B`.

If the block index `CAESAR_I` is outside the range $0..N-1$ (where N is the number of blocks in the system), the result is undefined.

.....

```
void CAESAR_ITERATE_STABLE_VARIABLE_SOLVE_1 (CAESAR_B, CAESAR_I, CAESAR_V,
                                             CAESAR_VALUE, CAESAR_LOOP)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   CAESAR_TYPE_POINTER CAESAR_V;
   CAESAR_TYPE_BOOLEAN *CAESAR_VALUE;
   void (*CAESAR_LOOP) (CAESAR_TYPE_SOLVE_1, CAESAR_TYPE_NATURAL,
      CAESAR_TYPE_POINTER, CAESAR_TYPE_BOOLEAN *);
   { ... }
```

This procedure provides an iterator which enumerates the boolean variables defined in the equation block of index `CAESAR_I` of the boolean equation system pointed to by `CAESAR_B` which are stable, i.e., whose value was computed by calls to the function `CAESAR_COMPUTE_SOLVE_1()` (see above). Only the variables computed since the last call of this procedure (or, in the case of the first call of this procedure, since the creation of the boolean equation system pointed to by `CAESAR_B`) are enumerated. At each iteration, `*CAESAR_VALUE` and the boolean variable pointed to by `CAESAR_V` are assigned a new value, such that `CAESAR_V` is defined in the block of index `CAESAR_I` of the boolean equation system pointed to by `CAESAR_B` and the value computed for `CAESAR_V` is equal to `*CAESAR_VALUE`. At each iteration, the procedure pointed to by `CAESAR_LOOP` is invoked, with the following parameters:

```
(*CAESAR_LOOP) (CAESAR_B, CAESAR_I, CAESAR_V, CAESAR_VALUE)
```

Therefore, any actual parameter supplied for the formal parameter `CAESAR_LOOP` must be a pointer to a procedure `caesar_p` whose declaration has the following form:

```
void caesar_p (caesar_bes, caesar_block, caesar_variable, caesar_value)
   CAESAR_TYPE_SOLVE_1 caesar_bes;
   CAESAR_TYPE_NATURAL caesar_block;
```

```
        CAESAR_TYPE_POINTER caesar_variable;
        CAESAR_TYPE_BOOLEAN *caesar_value;
        { ... }
```

Note: Parameters `CAESAR_V` and `CAESAR_VALUE` must point to (distinct) memory locations allocated before procedure `CAESAR_ITERATE_STABLE_VARIABLE_SOLVE_1()` is invoked. In no event will `CAESAR_ITERATE_STABLE_VARIABLE_SOLVE_1()` and `CAESAR_LOOP()` allocate memory for storing `CAESAR_V` and `CAESAR_VALUE`.

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

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

Note: The code that implements `CAESAR_ITERATE_STABLE_VARIABLE_SOLVE_1()` in the current version of the "solve_1" library is not reentrant, meaning that nested iterations will not work properly. This implies that any actual procedure `caesar_p` passed as value for formal parameter `CAESAR_LOOP` must not call (directly, nor transitively) `CAESAR_ITERATE_STABLE_VARIABLE_SOLVE_1()`.

Note: When invoked to solve a variable of interest, the function `CAESAR_COMPUTE_SOLVE_1()` usually explores and solves other variables upon which the variable of interest depends. This implies that the set of variables enumerated by a call to `CAESAR_ITERATE_STABLE_VARIABLE_SOLVE_1()` is usually larger than the set of variables defined in the block of index `CAESAR_I` of the system pointed to by `CAESAR_B` on which `CAESAR_COMPUTE_SOLVE_1()` was invoked since the last call of `CAESAR_ITERATE_STABLE_VARIABLE_SOLVE_1()`.

If the block index `CAESAR_I` is outside the range $0..N-1$ (where N is the number of blocks in the system), the effect is undefined.

.....

```
void CAESAR_START_DIAGNOSTIC_SOLVE_1 (CAESAR_B, CAESAR_I, CAESAR_V,
                                      CAESAR_MINIMAL, CAESAR_P)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   CAESAR_TYPE_POINTER CAESAR_V;
   CAESAR_TYPE_BOOLEAN CAESAR_MINIMAL;
   CAESAR_TYPE_POINTER *CAESAR_P;
   { ... }
```

This procedure initializes the diagnostic generation for the boolean variable pointed to by `CAESAR_V`, which must be defined in the equation block of index `CAESAR_I` of the boolean equation system pointed to by `CAESAR_B`. It must be called before starting to explore the diagnostic for the boolean variable using the procedure `CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1()` (see below).

Diagnostic information is computed by the resolution routines and kept in the internal tables associated to the blocks. Therefore, diagnostics can be generated only for boolean variables that were already solved by calls to `CAESAR_COMPUTE_SOLVE_1()` (see above). If the boolean variable pointed to by `CAESAR_V` was previously solved, the address of this boolean variable, which was stored in the internal table associated to the block of index `CAESAR_I`, is assigned to `*CAESAR_P`. If the boolean variable pointed to by `CAESAR_V` was not previously solved or a memory allocation failed during diagnostic recomputation (see below), the `NULL` value is assigned to `*CAESAR_P`.

Note: The memory area pointed to by *CAESAR_P must neither be modified, nor freed by the application program.

The value of CAESAR_MINIMAL influences the depth of the diagnostic (i.e., the length of the longest sequence without repeated vertices contained in the diagnostic) that will be generated for the boolean variable pointed to by CAESAR_V. If the value of CAESAR_MINIMAL is CAESAR_TRUE, then the diagnostic of the boolean variable will be recomputed in order to reduce its depth. If the value of CAESAR_MINIMAL is CAESAR_FALSE, the diagnostic of the boolean variable will be left unchanged, i.e., as it was computed when the variable was solved.

Note: Setting the value of CAESAR_MINIMAL to CAESAR_TRUE usually increases diagnostic generation time, especially if the variable pointed to by CAESAR_V was solved using a resolution mode based on depth-first search, such as resolution modes 0, 2, 3, and 4 (see the procedure CAESAR_CREATE_SOLVE_1() above).

If the block index CAESAR_I is outside the range $0..N-1$ (where N is the number of blocks in the system), the effect is undefined.

.....

```
void CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1 (CAESAR_B, CAESAR_I1, CAESAR_V1,
                                        CAESAR_L, CAESAR_I2, CAESAR_V2,
                                        CAESAR_LOOP)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I1;
   CAESAR_TYPE_POINTER CAESAR_V1;
   CAESAR_TYPE_POINTER *CAESAR_L;
   CAESAR_TYPE_NATURAL *CAESAR_I2;
   CAESAR_TYPE_POINTER *CAESAR_V2;
   void (*CAESAR_LOOP) (CAESAR_TYPE_SOLVE_1, CAESAR_TYPE_NATURAL,
      CAESAR_TYPE_POINTER, CAESAR_TYPE_POINTER *,
      CAESAR_TYPE_NATURAL *, CAESAR_TYPE_POINTER *);
   { ... }
```

This procedure provides an iterator which enumerates the successors of the boolean variable pointed to by CAESAR_V1 that are contained in the diagnostic of this variable. The variable pointed to by CAESAR_V1 must be defined in the equation block of index CAESAR_I1 of the boolean equation system pointed to by CAESAR_B. At each iteration, *CAESAR_I2 and *CAESAR_V2 are respectively assigned a block index and a pointer to a boolean variable such that "(CAESAR_V1, *CAESAR_V2)" is an edge of the diagnostic computed for the boolean variable pointed to by CAESAR_V1. The boolean variable pointed to by *CAESAR_V2 is defined in the equation block of index *CAESAR_I2 and is stored in the internal table associated to that block. Also, *CAESAR_L is assigned the information attached to the edge "(CAESAR_V1, *CAESAR_V2)" by the procedure pointed to by caesar_loop invoked by the iterator procedure caesar_p given as value for formal parameter CAESAR_VARIABLE_ITERATE when the boolean equation system pointed to by CAESAR_B was created (see procedure CAESAR_CREATE_SOLVE_1() above).

Note: Parameter CAESAR_V1 must contain the address of a boolean variable already stored in the internal table associated to the equation block of index CAESAR_I1. Such addresses of boolean variables are obtained as values assigned to the *CAESAR_P parameter of the CAESAR_START_DIAGNOSTIC_SOLVE_1() procedure (see above) or to the *CAESAR_V2 parameter of the CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1() procedure. If parameter CAESAR_V1 does not meet

this condition, the effect is undefined.

Note: The memory area pointed to by `*CAESAR_V2` should neither be modified, nor freed by the application program.

Note: The memory area pointed to by `*CAESAR_L` is entirely managed by the application program. The resolution algorithms manipulate the address of this memory area only by copying it and possibly by comparing it to `NULL` (see also procedure `CAESAR_CREATE_SOLVE_1()` above).

At each iteration, the procedure pointed to by `CAESAR_LOOP` is invoked, with the following parameters:

```
(*CAESAR_LOOP) (CAESAR_B, CAESAR_I1, CAESAR_V1,
                CAESAR_L, CAESAR_I2, CAESAR_V2)
```

Therefore, any actual parameter supplied for the formal parameter `CAESAR_LOOP` must be a pointer to a procedure `caesar_p` whose declaration has the following form:

```
void caesar_p (caesar_bes, caesar_block_1, caesar_variable_1,
               caesar_label, caesar_block_2, caesar_variable_2)
   CAESAR_TYPE_SOLVE_1 caesar_bes;
   CAESAR_TYPE_NATURAL caesar_block_1;
   CAESAR_TYPE_POINTER caesar_variable_1;
   CAESAR_TYPE_POINTER *caesar_label;
   CAESAR_TYPE_NATURAL *caesar_block_2;
   CAESAR_TYPE_POINTER *caesar_variable_2;
   { ... }
```

Note: Parameters `CAESAR_I2` and `CAESAR_V2` must point to (distinct) memory locations allocated before procedure `CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1()` is invoked. In no event will `CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1()` and `CAESAR_LOOP()` allocate memory for storing `CAESAR_I2` and `CAESAR_V2`.

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

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

Note: The code that implements `CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1()` in the current version of the "solve_1" library is not reentrant, meaning that nested iterations will not work properly. This implies that any actual procedure `caesar_p` passed as value for formal parameter `CAESAR_LOOP` must not call (directly, nor transitively) `CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1()`.

Additionally, this procedure sets two fields `caesar_creation` and `caesar_truncation` of type `CAESAR_TYPE_NATURAL` associated to the equation block of index `CAESAR_I1`. After any call to `CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1()`, these fields can be inspected using the two functions `CAESAR_CREATION_DIAGNOSTIC_SOLVE_1()` and `CAESAR_TRUNCATION_DIAGNOSTIC_SOLVE_1()` (see below). The values of these fields are set as follows:

- If the computation normally succeeds, then `caesar_creation` is set to the number of successors of the variable pointed to by `CAESAR_V1` that are contained in the diagnostic of this variable and `caesar_truncation` is set to zero.

- If allocation fails when enumerating the successors (due to a lack of memory), only a subset of the successors is enumerated. Then `caesar_creation` is set to the number of successors enumerated and `caesar_truncation` is set to the number of successors that have not been enumerated (this number is greater than zero).

If the block index `CAESAR_I1` is outside the range $0..N-1$ (where N is the number of blocks in the system), the effect is undefined.

.....

```
CAESAR_TYPE_NATURAL CAESAR_CREATION_DIAGNOSTIC_SOLVE_1 (CAESAR_B, CAESAR_I)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   { ... }
```

This function returns the value of the field `caesar_creation` associated to the equation block of index `CAESAR_I` of the boolean equation system pointed to by `CAESAR_B`, that was computed during the last call to `CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1()` (see above). This field can only be inspected using this function.

If the block index `CAESAR_I` is outside the range $0..N-1$ (where N is the number of blocks in the system), the effect is undefined.

.....

```
CAESAR_TYPE_NATURAL CAESAR_TRUNCATION_DIAGNOSTIC_SOLVE_1 (CAESAR_B, CAESAR_I)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   { ... }
```

This function returns the value of the field `caesar_truncation` associated to the equation block of index `CAESAR_I` of the boolean equation system pointed to by `CAESAR_B`, that was computed during the last call to `CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1()` (see above). This field can only be inspected using this function.

If the block index `CAESAR_I` is outside the range $0..N-1$ (where N is the number of blocks in the system), the effect is undefined.

.....

```
void CAESAR_READ_SOLVE_1 (CAESAR_B, CAESAR_FILE,
                          CAESAR_BLOCK_UNIQUE_RESOLUTION,
                          CAESAR_BLOCK_SOLVE_MODE)
   CAESAR_TYPE_SOLVE_1 *CAESAR_B;
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_BOOLEAN_FUNCTION_SOLVE_1 CAESAR_BLOCK_UNIQUE_RESOLUTION;
   CAESAR_TYPE_NATURAL_FUNCTION_SOLVE_1 CAESAR_BLOCK_SOLVE_MODE;
   { ... }
```

This procedure allocates a boolean equation system using `CAESAR_CREATE_SOLVE_1()` and assigns its address to `*CAESAR_B`. If the allocation fails, the `NULL` value is assigned to `*CAESAR_B`.

The value of `CAESAR_FILE` determines the file from which the boolean equation system (represented in textual form) will be read. Before this procedure is called, `CAESAR_FILE` must have been properly opened, for instance using `fopen(3)`.

The boolean equation system file is parsed: its contents is analyzed and stored into the boolean equation system `*CAESAR_B`.

So doing, various error conditions may occur: `CAESAR_FILE` is empty or has syntax errors; it has semantic errors (such as block or variable indexes out of range), etc. In such case, a detailed error message is displayed using the `CAESAR_WARNING()` procedure, and the `NULL` value is assigned to `*CAESAR_B`.

The actual values of the two remaining formal parameters will be stored and associated to the boolean equation system pointed to by `*CAESAR_B`.

The value of `CAESAR_BLOCK_UNIQUE_RESOLUTION` determines, for each equation block, whether only one variable (or several variables) of the block will be solved. If the value of `CAESAR_BLOCK_UNIQUE_RESOLUTION` is different from `NULL`, it will be given to the corresponding parameter in the call to `CAESAR_CREATE_SOLVE_1()` used to create the boolean equation system pointed to by `*CAESAR_B`. If the value of `CAESAR_BLOCK_UNIQUE_RESOLUTION` is `NULL`, then the value of the corresponding parameter in the call to `CAESAR_CREATE_SOLVE_1()` will be determined by the contents of `CAESAR_FILE`.

The value of `CAESAR_BLOCK_SOLVE_MODE` determines, for each equation block, its corresponding resolution mode, determining which algorithm will be used by the resolution routine associated to the block. If the value of `CAESAR_BLOCK_SOLVE_MODE` is different from `NULL`, it will be given to the corresponding parameter in the call to `CAESAR_CREATE_SOLVE_1()` used to create the boolean equation system pointed to by `*CAESAR_B`. If the value of `CAESAR_BLOCK_SOLVE_MODE` is `NULL`, then the value of the corresponding parameter in the call to `CAESAR_CREATE_SOLVE_1()` will be determined by the contents of `CAESAR_FILE`.

Note: These two parameters allow to overwrite the values of the corresponding parameters of `CAESAR_CREATE_SOLVE_1()` determined by the contents of `CAESAR_FILE`. This is useful for applying last minute changes on the resolution of the boolean equation system read from `CAESAR_FILE`.

The contents of the boolean variables defined in the equation blocks of the system pointed to by `*CAESAR_B` are natural numbers, i.e., values of type `CAESAR_TYPE_NATURAL`. Each variable `Xi` defined by an equation of the system is represented by the value `i` of its index. As an example, the following portion of C code implements the resolution of a boolean equation system created using `CAESAR_READ_SOLVE_1()`:

```
CAESAR_TYPE_SOLVE_1 caesar_bes;
CAESAR_TYPE_FILE caesar_file;
CAESAR_TYPE_NATURAL caesar_variable;
CAESAR_TYPE_BOOLEAN caesar_value;

if ((caesar_file = fopen ("file.bes", "r")) != NULL) {
   CAESAR_READ_SOLVE_1 (&caesar_bes, caesar_file, NULL, NULL);
   if (caesar_bes != NULL) {
      /* resolution of variable 0 defined in the block of index 0 */
      caesar_variable = 0;
      caesar_value = CAESAR_COMPUTE_SOLVE_1 (caesar_bes, 0,
                        (CAESAR_TYPE_POINTER) (&caesar_variable));
   }
}
```

.....

```
void CAESAR_WRITE_SOLVE_1 (CAESAR_B, CAESAR_I, CAESAR_V,
                           CAESAR_FILE, CAESAR_DIAGNOSTIC)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_NATURAL CAESAR_I;
   CAESAR_TYPE_POINTER CAESAR_V;
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_BOOLEAN CAESAR_DIAGNOSTIC;
   { ... }
```

This procedure writes a portion of the boolean equation system pointed to by `CAESAR_B` in textual form into the file `CAESAR_FILE`. The portion written contains equations defining boolean variables upon which the boolean variable pointed to by `CAESAR_V`, which must be defined in the equation block of index `CAESAR_I`, depends either directly, or transitively.

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

So doing, various error conditions may occur: `CAESAR_FILE` is not writable; a memory allocation failed, etc. In such case, a detailed error message is displayed using the `CAESAR_WARNING()` procedure and the portion of boolean equation system written into `CAESAR_FILE` may be truncated.

If the block index `CAESAR_I` is outside the range $0..N-1$ (where N is the number of blocks in the system), the effect is undefined.

If the value of `CAESAR_DIAGNOSTIC` is equal to `CAESAR_FALSE`, the portion of boolean equation system written into `CAESAR_FILE` will contain all equations defining the boolean variables upon which the boolean variable pointed to by `CAESAR_V` depends. Otherwise, this portion will contain only the equations defining the variables contained in the diagnostic for the variable pointed to by `CAESAR_V`, which must have been solved previously by using `CAESAR_COMPUTE_SOLVE_1()`; if this is not the case, an error message is displayed using the `CAESAR_WARNING()` procedure and nothing is written into `CAESAR_FILE`.

Note: This procedure does not perform the resolution of the variable pointed to by `CAESAR_V`.

.....

```
CAESAR_FORMAT CAESAR_FORMAT_SOLVE_1 (CAESAR_B, CAESAR_FORMAT)
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the boolean equation system pointed to by `CAESAR_B` will be printed by the procedure `CAESAR_PRINT_SOLVE_1()` (see below). Currently, the following formats are available:

- With format 0, statistical information concerning the boolean equation system is displayed such as: the size of variables and resolution modes for each equation block of the system, the number of boolean variables explored during resolution, etc.
- With format 1, statistical information concerning the internal tables associated to the equation blocks of the system is printed using the procedure `CAESAR_PRINT_TABLE_1()`.
- With format 2, the contents of the internal tables associated to the equation blocks of the system are printed using the procedure `CAESAR_PRINT_TABLE_1()`.

- (no other format available yet)

By default, the current format of each boolean equation system is initialized to 0.

When called with `CAESAR_FORMAT` between 0 and 2, this fonction sets the current format of `CAESAR_B` 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_B` 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_B`. If `CAESAR_FORMAT` is equal to the constant `CAESAR_MAXIMAL_FORMAT`, the result is the maximal format value (i.e., 2). In all other cases, the effect of this function is undefined.

.....

```
CAESAR_TYPE_FORMAT CAESAR_MAX_FORMAT_SOLVE_1 ()
   { ... }
```

Caution! This function is deprecated. It should no longer be used, as it might be removed from future versions of the OPEN/CÆSAR. Use function `CAESAR_FORMAT_SOLVE_1()` instead, called with argument `CAESAR_MAXIMAL_FORMAT`.

This function returns the maximal format value available for printing boolean equation systems.

.....

```
void CAESAR_PRINT_SOLVE_1 (CAESAR_FILE, CAESAR_B)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_SOLVE_1 CAESAR_B;
   { ... }
```

This procedure prints on file `CAESAR_FILE` an ASCII text containing various informations about the boolean equation system pointed to by `CAESAR_B`. The nature of these informations is determined by the current format of the boolean equation system pointed to by `CAESAR_B`.

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

.....

17.11 Bibliography

[Mat03] Radu Mateescu. A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems. In Hubert Garavel and John Hatcliff, editors, Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland), Lecture Notes in Computer Science vol. 2619, pages 81–96. Springer Verlag, April 2003. Available from `http://cadp.inria.fr/publications/Mateescu-03-a.html`

[Mat06] Radu Mateescu. CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution

of Alternation-Free Boolean Equation Systems. Springer International Journal on Software Tools for Technology Transfer (STTT), 8(1):37–56, February 2006. Available from `http://cadp.inria.fr/publications/Mateescu-06-a.html`

Chapter 18

The "solve_2" library (version 1.1)

by Radu Mateescu

18.1 Purpose

The "solve_2" library provides primitives for solving linear equation systems, which are provided "on the fly". This library can be used as a back-end for various on the fly verification tools that formulate their corresponding problems (e.g., probabilistic verification) in terms of linear equation systems.

18.2 Usage

The "solve_2" library consists of:

- a predefined header file "`caesar_solve_2.h`";
- the precompiled library file "`libcaesar.a`", which implements the features described in "`caesar_solve_2.h`".

Note: The "solve_2" library is a software layer built above the primitives offered by the "standard", "area_1", and "table_1" libraries, and by the OPEN/CÆSAR graph module.

18.3 Linear equation systems

A "linear equation system" is a set of (possibly recursive) equations with numerical variables on their left-hand sides and linear algebraic expressions on their right-hand sides.

The expressions on the right-hand sides of equations are built from numerical variables (`X1`, `X2`, etc.) and real numbers (`3.1416`, `-0.5`, etc.) using arithmetic operators (`+`, `*`).

Equations are of the following form:

```
Xi = Ci1 * X1 + ... + Cin * Xn + Bi
```

where the real numbers `Cij` are the coefficients of variables `Xj` and the real numbers `Bi` are constants, for i, j between 1 and n. If a coefficient `Cij` is equal to 0, the term `Cij * Xj` is omitted.

A variable is defined in a linear equation system if it occurs as left-hand side in an equation of the system. Multiple definitions of a variable are forbidden, i.e., a variable can occur as left-hand side in at most one equation of a linear equation system. Each variable occurring on the right-hand side of an equation in a linear equation system must be defined by some equation of the system.

If the formula on the right-hand side of the equation defining a variable `Xi` contains another variable `Xj`, there is a dependency from `Xi` to `Xj`.

An example of a linear equation system containing four equations is shown below.

```
X1 = 0.4 * X2 + 0.6 * X3
X2 = 0.5 * X1 + 0.2
X3 = 0.3
```

This system contains a cyclic dependency between variables `X1` and `X2`.

18.4 On the fly resolution

Given a linear equation system, the on the fly (or local) resolution problem consists in computing the value of a particular variable defined in the system. Contrary to global resolution, which consists in computing the values of all variables defined in the system (and therefore must examine all equations of the system), on the fly resolution computes the value of a variable without necessarily examining all equations of the system. This resolution technique allows to construct the linear equation system in a demand-driven manner, and hence it is useful for building tools for on the fly verification, which explore one or more labelled transition systems incrementally.

The on the fly resolution method for linear equation systems implemented in the "solve_2" library (see [MR18]) proceeds as follows. Each system has an associated resolution routine responsible for computing the values of the variables defined in the system.

The resolution routine associated to a linear equation system is easier to develop using a representation of the system as a dependency graph (also known as Signal Flow Graph), which provides a more intuitive view of the dependencies between variables. Given an equation system, its corresponding dependency graph is defined as follows:

- For each variable `Xi` occurring in the system, there is a vertex `Xi` in the dependency graph.
- For each dependency from a variable `Xi` to a variable `Xj` such that the term `Cij * Xj` occurs on the right-hand side of the equation defining `Xi`, there is an edge "(`Xi`, `Cij`, `Xj`)" in the dependency graph.
- There is a special sink vertex `X0` (i.e., without any outgoing transitions), which denotes the constant `1.0`.

To construct its dependency graph, the linear equation system shown above must be first transformed into the equivalent form below, by replacing each constant `Bi` by a term `Bi * X0` and adding the equation `X0 = 1.0`:

```
X0 = 1.0
X1 = 0.4 * X2 + 0.6 * X3
X2 = 0.5 * X1 + 0.2 * X0
X3 = 0.3 * X0
```

Its dependency graph is the following:

```
(X1, 0.4, X2)
(X1, 0.6, X3)
(X2, 0.5, X1)
(X2, 0.2, X0)
(X3, 0.3, X0)
```

The resolution routine associated to a linear equation system will explore forward the corresponding dependency graph and will propagate backward the values of variables already stabilized (i.e., the value of which has been determined). When solving a variable, only the part of the dependency graph relevant for computing the value of the variable is explored. For example, solving the variable `X3` of the system above requires to explore only the dependency "(`X3`, `0.3`, `X0`)", yielding the value `0.3` for `X3`. The resolution of variables `X1` and `X2`, which are present on a cycle of the dependency graph, is carried out by a traditional linear resolution algorithm (e.g., an iterative algorithm) executed on the equations defining `X1` and `X2`, yielding the values 0.325 and 0.3625, respectively.

To enable on the fly exploration, the dependency graphs associated to linear equation systems are represented in a generic, implicit manner using a scheme similar to the one defined by the OPEN/CÆSAR graph module for representing labelled transition systems. This representation roughly consists of the following ingredients (see procedure `CAESAR_CREATE_SOLVE_2()` below for additional details):

- Variables (vertices of the dependency graph) are represented as pointers to memory areas of fixed size. The precise meaning of the variable contents is defined by the application program and is not relevant for the resolution algorithms.
- A linear equation system is equipped with several functions computing various information about the variables defined in the system: a comparison function, a hashing function, a printing function, and an iterator procedure which enumerates the successors of a variable in the dependency graph.

To speed up the resolution process, a linear equation system has associated an internal table which stores the variables already explored during the previous calls of the resolution routine associated to the system. This avoids recomputations of variables by subsequent calls of the resolution routine, leading (for the sparse linear equation systems encountered in verification problems) to an overall resolution process of time complexity linear in the size of the system (number of variables and operators).

18.5 Description

The "solve_2" library allows to create and handle linear equation systems on the fly, providing procedures for resolution, inspection, and writing information to text files.

18.6 Features

```
typedef CAESAR_TYPE_ABSTRACT (...) CAESAR_TYPE_SOLVE_2;
```

This type denotes a pointer to the concrete representation of a linear equation system. This representation is supposed to be "opaque".

. . . . . .

```
typedef double CAESAR_TYPE_REAL;
```

`CAESAR_TYPE_REAL` is the real number type used in the "solve_2" library.

.....

```
typedef enum {
     CAESAR_NONE_SOLVE_2,
     CAESAR_MULTIPLE_RESOLUTION_SOLVE_2,
     CAESAR_MEMORY_SHORTAGE_SOLVE_2,
     CAESAR_SINGULAR_SOLVE_2
}    CAESAR_TYPE_ERROR_SOLVE_2;
```

This enumerated type defines the error codes produced as a side effect by calls to the function `CAESAR_COMPUTE_SOLVE_2()` (see below), which performs the resolution of a variable defined in a linear equation system. The error codes have the following meaning:

- `CAESAR_NONE_SOLVE_2` indicates that the resolution was performed successfully.
- `CAESAR_MULTIPLE_RESOLUTION_SOLVE_2` indicates that another resolution of a variable of the system was already performed, whereas at the creation of the linear equation system (see procedure `CAESAR_CREATE_SOLVE_2()` below) a single resolution was specified for the system.
- `CAESAR_MEMORY_SHORTAGE_SOLVE_2` indicates that a memory allocation failed during the resolution.
- `CAESAR_SINGULAR_SOLVE_2` indicates that the linear equation system contains singularities, i.e., either it has no solution, or it does not have a unique solution.

Note: The error code produced by a call to `CAESAR_COMPUTE_SOLVE_2()` can be obtained by using the function `CAESAR_STATUS_COMPUTE_SOLVE_2()` (see below).

.....

```
void CAESAR_CREATE_SOLVE_2 (CAESAR_L,
                            CAESAR_UNIQUE_RESOLUTION,
                            CAESAR_SOLVE_MODE,
                            CAESAR_EPSILON,
                            CAESAR_VARIABLE_AREA,
                            CAESAR_LIMIT_SIZE,
                            CAESAR_HASH_SIZE,
                            CAESAR_PRIME,
                            CAESAR_VARIABLE_COMPARE,
                            CAESAR_VARIABLE_HASH,
                            CAESAR_VARIABLE_PRINT,
                            CAESAR_VARIABLE_ITERATE,
                            CAESAR_INFO)
   CAESAR_TYPE_SOLVE_2 *CAESAR_L;
   CAESAR_TYPE_BOOLEAN CAESAR_UNIQUE_RESOLUTION;
   CAESAR_TYPE_NATURAL CAESAR_SOLVE_MODE;
```

```
CAESAR_TYPE_REAL CAESAR_EPSILON;
CAESAR_TYPE_AREA_1 CAESAR_VARIABLE_AREA;
CAESAR_TYPE_NATURAL CAESAR_LIMIT_SIZE;
CAESAR_TYPE_NATURAL CAESAR_HASH_SIZE;
CAESAR_TYPE_BOOLEAN CAESAR_PRIME;
CAESAR_TYPE_COMPARE_FUNCTION CAESAR_VARIABLE_COMPARE;
CAESAR_TYPE_HASH_FUNCTION CAESAR_VARIABLE_HASH;
CAESAR_TYPE_PRINT_FUNCTION CAESAR_VARIABLE_PRINT;
void (*CAESAR_VARIABLE_ITERATE) (CAESAR_TYPE_POINTER, CAESAR_TYPE_POINTER,
   void (*) (CAESAR_TYPE_REAL, CAESAR_TYPE_POINTER));
CAESAR_TYPE_POINTER CAESAR_INFO;
{ ... }
```

This procedure allocates a linear equation system 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_SOLVE_2` is a pointer type, any variable `CAESAR_L` of type `CAESAR_TYPE_SOLVE_2` must be allocated before used, for instance using:

```
CAESAR_CREATE_SOLVE_2 (&CAESAR_L, ...);
```

The actual value of the formal parameter `CAESAR_UNIQUE_RESOLUTION` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used to assign a boolean indicating whether only one variable (or several variables) of the system will be solved.

The actual value of the formal parameter `CAESAR_SOLVE_MODE` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used to assign the resolution mode, determining which algorithm will be used by the resolution routine associated to the system.

Currently, the following resolution modes are available:

- Mode 0 corresponds to a resolution algorithm based upon a depth-first search of the dependency graph associated to the system, with detection of strongly connected components (SCCs) and iterative resolution of the variables contained in the SCCs [MR18]. This mode works only for stochastic linear equation systems, whose equations `Xi` = `Ci1 * X1` + ... + `Cin * Xn` + `Bi` are such that all coefficients `Cij` and constant `Bi` are positive, and their sum `Ci1` + ... + `Cin` + `Bi` is less than or equal to 1.0.
- (no other resolution mode available yet)

If the resolution mode given by `CAESAR_SOLVE_MODE` is not among the resolution modes above, or the linear equation system does not have the form expected by that resolution mode, the effect is undefined.

The actual value of the formal parameter `CAESAR_EPSILON` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used to assign the precision of comparisons between real numbers during the resolution. If this value is equal to zero, it is replaced by the default value `1E-6`.

The actual value of the formal parameter `CAESAR_VARIABLE_AREA` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used to assign the (constant) size and (constant) alignment factor of the variables defined in the system.

The actual value of the formal parameter `CAESAR_LIMIT_SIZE` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used to assign the maximal size (number of

items) of the internal table associated to the system. This value must be less or equal to a predefined value M (see the "table_1" library). If it is equal to zero, it is replaced by the default value M.

The actual value of the formal parameter `CAESAR_HASH_SIZE` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used to assign the size (number of entries) of the hash-table accompanying the internal table associated to the system. If this value is equal to zero, it is replaced by a default value greater than zero (see the "table_1" library).

The actual value of the formal parameter `CAESAR_PRIME` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used to assign a boolean value allowing to adjust the size of the hash-table accompanying the internal table associated to the system. If this value is equal to `CAESAR_TRUE` and if the value of `CAESAR_HASH_SIZE` is not a prime number, this value will be replaced by the nearest smaller prime number (since some hash functions require prime modulus). Otherwise, the value of `CAESAR_HASH_SIZE` will be kept unchanged (see the "table_1" library).

The actual value of the formal parameter `CAESAR_VARIABLE_COMPARE` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used as a comparison function for the variables defined in the system.

Precisely, the actual value of `CAESAR_VARIABLE_COMPARE` should be a pointer to a comparison function `caesar_f` with two parameters `caesar_variable_1` and `caesar_variable_2` that returns `CAESAR_TRUE` (resp. `CAESAR_FALSE`) if the variables pointed to by `caesar_variable_1` and `caesar_variable_2` are equal (resp. different). A pointer to the linear equation system (i.e., the value assigned to `*CAESAR_B`) can be obtained within `caesar_f` by calling the function `CAESAR_CURRENT_SYSTEM_SOLVE_1()` (see below).

The actual value of the formal parameter `CAESAR_VARIABLE_HASH` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used as a hash-function for the variables defined in the system.

Precisely, the actual value of `CAESAR_VARIABLE_HASH` should be a pointer to a hash function `caesar_f` with two parameters `caesar_variable` and `caesar_modulus` that returns a hash-value computed on the byte string `caesar_variable [0]` up to `caesar_variable [caesar_size - 1]`, where the actual value of `caesar_size` will always be equal to the size of the variable pointed to by `caesar_variable`. This hash-value must belong to the range 0..`caesar_modulus` $-$ 1. A pointer to the linear equation system (i.e., the value assigned to `*CAESAR_L`) can be obtained within `caesar_f` by calling the function `CAESAR_CURRENT_SYSTEM_SOLVE_2()` (see below).

The actual value of the formal parameter `CAESAR_VARIABLE_PRINT` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used as a printing procedure for the variables defined in the system.

Precisely, the actual value of `CAESAR_VARIABLE_PRINT` should be a pointer to a printing procedure `caesar_p` with two parameters `caesar_file` and `caesar_variable` that prints to file `caesar_file` information about the contents of the variable pointed to by `caesar_variable`. A pointer to the linear equation system (i.e., the value assigned to `*CAESAR_L`) can be obtained within `caesar_p` by calling the function `CAESAR_CURRENT_SYSTEM_SOLVE_2()` (see below).

The actual value of the formal parameter `CAESAR_VARIABLE_ITERATE` will be stored and associated to the linear equation system pointed to by `*CAESAR_L`. It will be used as an iterator procedure enumerating all successors of the variables defined in the system.

Any user-defined procedure `caesar_p` can be used as an actual value for formal parameter `CAESAR_VARIABLE_ITERATE`, provided that its declaration has the form:

```
void caesar_p (caesar_variable_1, caesar_variable_2, caesar_loop)
   CAESAR_TYPE_POINTER caesar_variable_1;
```

```
CAESAR_TYPE_POINTER caesar_variable_2;
void (*caesar_loop) (CAESAR_TYPE_REAL, CAESAR_TYPE_POINTER);
{ ... }
```

This procedure `caesar_p` enumerates all successors of the variable pointed to by `caesar_variable_1`. A pointer to the linear equation system (i.e., the value assigned to `*CAESAR_L`) can be obtained within `caesar_p` by calling the function `CAESAR_CURRENT_SYSTEM_SOLVE_2()` (see below). At each iteration performed by `caesar_p`, two actions must be carried out:

- First, the variable pointed to by `caesar_variable_2` must be assigned a new value, such that `caesar_variable_1` and `caesar_variable_2` are the source and target vertices of an edge in the dependency graph.
- Second, the procedure pointed to by `caesar_loop` must be called. The actual value of the formal parameter `caesar_loop` is a procedure `caesar_q` whose declaration has the form:

  ```
  void caesar_q (caesar_coefficient_2, caesar_variable_2)
     CAESAR_TYPE_REAL caesar_coefficient_2;
     CAESAR_TYPE_POINTER caesar_variable_2;
     { ... }
  ```

  Therefore, each call to the procedure pointed to by `caesar_loop` must have the following parameters:

  ```
  (*caesar_loop) (caesar_coefficient_2, caesar_variable_2)
  ```

  Parameter `caesar_coefficient_2` represents the coefficient of the variable `caesar_variable_2` on the right-hand side of the equation defining `caesar_variable_1`, i.e., "(`caesar_variable_1`, `caesar_coefficient_2`, `caesar_variable_2`)" represents an edge of the dependency graph.

Note: The memory area pointed to by the parameter `caesar_variable_1` contains a variable and should neither be modified, nor freed by the procedure `caesar_p`.

Note: The memory area pointed to by the parameter `caesar_variable_2` is already allocated and should not be freed by the procedure `caesar_p`.

The value of `CAESAR_INFO` has no effect on the execution of procedure `CAESAR_CREATE_SOLVE_2()`. Parameter `CAESAR_INFO` is intended to serve for future extensions of this procedure; when using the current version of the "solve_2" library, it is recommended to set this parameter to `NULL`.

.....

```
CAESAR_TYPE_SOLVE_2 CAESAR_CURRENT_SYSTEM_SOLVE_2 ()
   { ... }
```

This function returns a pointer to the linear equation system which is currently under resolution. It should be called only within the functions and procedures given as actual values for the formal parameters `CAESAR_VARIABLE_COMPARE`, `CAESAR_VARIABLE_HASH`, `CAESAR_VARIABLE_PRINT`, and `CAESAR_VARIABLE_ITERATE` of procedure `CAESAR_CREATE_SOLVE_2()` (see above); in this case, the result is a pointer to the linear equation system created by the call to `CAESAR_CREATE_SOLVE_2()`. If this function is called anywhere else in the application program, the result is undefined.

Note: This function allows to invoke, within the five aforementioned functions and procedures, various primitives of the "solve_2" library on the current linear equation system (e.g., resolution, printing, etc.).

. . . . . .

```
void CAESAR_DELETE_SOLVE_2 (CAESAR_L)
   CAESAR_TYPE_SOLVE_2 *CAESAR_L;
   { ... }
```

This procedure frees the memory space corresponding to the linear equation system pointed to by `*CAESAR_L` using `CAESAR_DELETE()`. The variables stored in the internal table allocated during previous resolutions (if any) of the linear equation system are also freed. Afterwards, the `NULL` value is assigned to `*CAESAR_L`.

. . . . . .

```
void CAESAR_PURGE_SOLVE_2 (CAESAR_L)
   CAESAR_TYPE_SOLVE_2 CAESAR_L;
   { ... }
```

This procedure reinitializes the information associated to the linear equation system pointed to by `CAESAR_L`. The internal table associated to the system is emptied using `CAESAR_PURGE_TABLE_1()`. Afterwards, the linear equation system is exactly in the same state as after its creation using `CAESAR_CREATE_SOLVE_2()`.

. . . . . .

```
CAESAR_TYPE_REAL CAESAR_COMPUTE_SOLVE_2 (CAESAR_L, CAESAR_V)
   CAESAR_TYPE_SOLVE_2 CAESAR_L;
   CAESAR_TYPE_POINTER CAESAR_V;
   { ... }
```

This function computes the value of the variable pointed to by `CAESAR_V`, which must be defined in the linear equation system pointed to by `CAESAR_L`. It also sets a field of type `CAESAR_TYPE_ERROR_SOLVE_2` associated to the system, indicating whether the resolution was carried out successfully or not; this field can be inspected using the function `CAESAR_STATUS_COMPUTE_SOLVE_2()` (see below).

. . . . . .

```
CAESAR_TYPE_ERROR_SOLVE_2 CAESAR_STATUS_COMPUTE_SOLVE_2 (CAESAR_L)
   CAESAR_TYPE_SOLVE_2 CAESAR_L;
   { ... }
```

This function returns the status of the last resolution performed by a call to the function `CAESAR_COMPUTE_SOLVE_2()` (see above) on a variable defined in the linear equation system pointed

to by CAESAR_L.

.....

```
CAESAR_FORMAT CAESAR_FORMAT_SOLVE_2 (CAESAR_L, CAESAR_FORMAT)
   CAESAR_TYPE_SOLVE_2 CAESAR_L;
   CAESAR_TYPE_FORMAT CAESAR_FORMAT;
   { ... }
```

This function allows to control the format under which the linear equation system pointed to by CAESAR_L will be printed by the procedure CAESAR_PRINT_SOLVE_2() (see below). Currently, the following formats are available:

- With format 0, statistical information concerning the linear equation system is displayed, such as: the size of variables and resolution mode, the number of variables explored during resolution, etc.
- With format 1, statistical information concerning the internal table associated to the system is printed using the procedure CAESAR_PRINT_TABLE_1().
- With format 2, the contents of the internal table associated to the system is printed using the procedure CAESAR_PRINT_TABLE_1().
- (no other format available yet)

By default, the current format of each linear equation system is initialized to 0.

When called with CAESAR_FORMAT between 0 and 2, this function sets the current format of CAESAR_L 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_L 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_L. If CAESAR_FORMAT is equal to the constant CAESAR_MAXIMAL_FORMAT, the result is the maximal format value (i.e., 2). In all other cases, the effect of this function is undefined.

.....

```
void CAESAR_PRINT_SOLVE_2 (CAESAR_FILE, CAESAR_L)
   CAESAR_TYPE_FILE CAESAR_FILE;
   CAESAR_TYPE_SOLVE_2 CAESAR_L;
   { ... }
```

This procedure prints on file CAESAR_FILE an ASCII text containing various informations about the linear equation system pointed to by CAESAR_L. The nature of these informations is determined by the current format of the linear equation system pointed to by CAESAR_L.

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

.....

18.7 Bibliography

[MR18] Radu Mateescu and Jose Ignacio Requeno. On-the-Fly Model Checking for Extended Action-Based Probabilistic Operators. Springer International Journal on Software Tools for Technology Transfer (STTT), 20(5):563–587, October 2018. Available from `http://hal.inria.fr/hal-01862754/en`

Index