NAME

caesar_solve_1 – the ‘‘solve_1’’ library of OPEN/CAESAR

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.

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/CAESAR* graph module.

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’’.

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).

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).

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)

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/CAESAR* 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).

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/CAESAR* 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).

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.

FEATURES

....

CAESAR_TYPE_SOLVE_1

```
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”.

....

CAESAR_TYPE_BLOCK_SIGN_SOLVE_1

```
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.

....

CAESAR_MINIMAL_FIXED_POINT_SOLVE_1

```
#define CAESAR_MINIMAL_FIXED_POINT_SOLVE_1 \
   ((CAESAR_TYPE_BLOCK_SIGN_SOLVE_1) CAESAR_TRUE)
```

This constant denotes the minimal fixed point sign.

....

CAESAR_MAXIMAL_FIXED_POINT_SOLVE_1

```
#define CAESAR_MAXIMAL_FIXED_POINT_SOLVE_1 \
   ((CAESAR_TYPE_BLOCK_SIGN_SOLVE_1) CAESAR_FALSE)
```

This constant denotes the maximal fixed point sign.

....

CAESAR_TYPE_VARIABLE_KIND_SOLVE_1

```
typedef CAESAR_TYPE_BOOLEAN CAESAR_TYPE_VARIABLE_KIND_SOLVE_1;
```

This type indicates the kind (disjunctive or conjunctive) associated to a boolean variable.

....

CAESAR_DISJUNCTIVE_VARIABLE_SOLVE_1

```
#define CAESAR_DISJUNCTIVE_VARIABLE_SOLVE_1 \
   ((CAESAR_TYPE_VARIABLE_KIND_SOLVE_1) CAESAR_TRUE)
```

This constant denotes the disjunctive variable kind.

....

CAESAR_CONJUNCTIVE_VARIABLE_SOLVE_1

```
#define CAESAR_CONJUNCTIVE_VARIABLE_SOLVE_1 \
   ((CAESAR_TYPE_VARIABLE_KIND_SOLVE_1) CAESAR_FALSE)
```

This constant denotes the conjunctive variable kind.

....

CAESAR_TYPE_BLOCK_SIGN_FUNCTION_SOLVE_1

```
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.

....

CAESAR_TYPE_VARIABLE_KIND_FUNCTION_SOLVE_1

```
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.

....

CAESAR_TYPE_AREA_FUNCTION_SOLVE_1

```
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.

....

CAESAR_TYPE_BOOLEAN_FUNCTION_SOLVE_1

```
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.

....

CAESAR_TYPE_NATURAL_FUNCTION_SOLVE_1

```
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.

....

CAESAR_TYPE_ERROR_SOLVE_1

```
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).

....

CAESAR_CREATE_SOLVE_1

```
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_CURRENT_SYSTEM_SOLVE_1

```
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_CURRENT_BLOCK_SOLVE_1

```
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).

....

CAESAR_DELETE_SOLVE_1

```
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**.

....

CAESAR_PURGE_BLOCK_SOLVE_1

```
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_COMPUTE_SOLVE_1

```
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_STATUS_COMPUTE_SOLVE_1

```
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.

....

CAESAR_ITERATE_STABLE_VARIABLE_SOLVE_1

```
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.

....

CAESAR_START_DIAGNOSTIC_SOLVE_1

```
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.

....

CAESAR_ITERATE_DIAGNOSTIC_SOLVE_1

```
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_CREATION_DIAGNOSTIC_SOLVE_1

```
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_TRUNCATION_DIAGNOSTIC_SOLVE_1

```
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.

....

CAESAR_READ_SOLVE_1

```
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));
   }
}
```

....

CAESAR_WRITE_SOLVE_1

```
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_SOLVE_1

```
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_MAX_FORMAT_SOLVE_1

```
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/CAESAR*. 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.

....

CAESAR_PRINT_SOLVE_1

```
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)`**.

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`**

....

AUTHOR(S)

Radu Mateescu

FILES

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

SEE ALSO

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

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

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

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

BUGS

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