Formal Specification and Verification of Fully Asynchronous Implementations of the Data Encryption Standard

Wendelin Serwe
Inria
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
wendelin.serwe@inria.fr

This paper presents two formal models of the Data Encryption Standard (DES), a first using the international standard LOTOS, and a second using the more recent process calculus LNT. Both models encode the DES in the style of asynchronous circuits, i.e., the data-flow blocks of the DES algorithm are represented by processes communicating via rendezvous. To ensure correctness of the models, several techniques have been applied, including model checking, equivalence checking, and comparing the results produced by a prototype automatically generated from the formal model with those of existing implementations of the DES. The complete code of the models is provided as appendices and also available on the website of the CADP verification toolbox.

1 Introduction

The Data Encryption Standard (DES) is a symmetric-key encryption algorithm, that has been for almost 30 years a Federal Information Processing Standard [17]. At present, the main weakness of the algorithm is its use of keys of 56 bits, which are too short to withstand brute force attacks. To address this issue, the Triple Data Encryption Algorithm (TDEA or Triple DES) variant [18] applies the DES algorithm three times (encryption, decryption, encryption), using three different keys, and is still an approved symmetric cryptographic algorithm for 8-byte block ciphers, e.g., for secure payment systems [4, Annex B1.1].

An interesting aspect of the DES is that it is specified by a data-flow diagram, i.e., as a set of blocks communicating by message passing. Such an architecture is naturally asynchronous because there is no need for a global clock synchronizing these various blocks. Due to different interleavings of the block executions, the exists a risk that the DES execution produces a nondeterministic result, although this is not expected. This problem naturally lends itself to analysis with process calculi.

A prior case-study [1] analyzed an asynchronous circuit\(^1\) implementing the DES. This circuit was specified in the CHP (Communicating Hardware Processes) [13] process calculus, which was translated into the IF [2] formalism, and subsequently verified using the CADP toolbox [8]\(^2\), i.e., the corresponding state space was generated as an LTS (Labeled Transition System) and several properties were verified by model and equivalence checking. To reduce the state space, this prior case-study replaced some of the parallelism by sequential compositions, removing some asynchronism.

The present paper goes further and analyzes the fully asynchronous DES, describing formal models of the DES in two different process calculi, namely the international standard LOTOS [12] and the

\(^1\) An asynchronous circuit is a circuit without clocks, where the different components of the circuit synchronize via handshake protocols. Implementing a cryptographic algorithm as an asynchronous circuit makes it more robust against side channel attacks based on the analysis of the power consumption or radio-emission, which have significantly fewer peaks that could be exploited to get information about the key or data being encrypted [15].

\(^2\) http://cadp.inria.fr
more recent LNT language [3], both supported by CADP. The LOTOS model of the DES was directly derived from the DES standard [17] to experiment with the application of LOTOS for the analysis of asynchronous circuits. In August 2015, the LOTOS model was rewritten into an equivalent LNT model to fulfill the expectations of the MARS workshop. Both models have been analyzed using different techniques, namely data abstraction, model checking, equivalence checking, and the automatic generation of a prototype software implementation. All these verification steps have been automated by an SVL (Script Verification Language) [6] script.

The rest of this paper is organized as follows. Section 2 briefly presents the data-flow architecture of the DES. Section 3 discusses modeling challenges and choices. Section 4 (respectively, 5) presents the steps undertaken to assess the correctness of the model with (respectively without) data abstraction. Section 6 gives concluding remarks. Appendix A is the complete source code of the LNT model. Appendix B is the complete source code of the LOTOS model. Appendix C contains the C code required to generate a prototype with the EXEC/CÆSAR framework. Appendix D provides the SVL script to execute the verification scenarios described in this paper.

2 Architecture of the Asynchronous Data Encryption Standard

The DES is an iterative algorithm, which takes as input a 64-bit word\(^3\) of data, a 64-bit key (from which only 56 bits are used), and a bit indicating whether the data is to be encrypted or decrypted, and produces as output a 64-bit word of encrypted or decrypted data. The DES first applies an initial permutation on the input data, splits the 64-bit data word into two 32-bit data words, named \(L_0\) and \(R_0\), and then iteratively computes \(L_{n+1} = R_n\) and \(R_{n+1} = L_n \oplus f(R_n, K_{n+1})\), where \(\oplus\) stands for the bit-wise sum, and \(f\) is the so-called cipher function. The final result is obtained as the inverse initial permutation applied to the concatenation of \(R_{16}\) and \(L_{16}\).

In each iteration, the cipher function \(f\) first applies a function named \(E\) to expand the 32-bit word \(R_n\) to a 48-bit word, and then computes the bit-wise sum with the 48-bit subkey \(K_n\). The result is split into eight 6-bit words, each of which is transformed by a so-called S-box into a 4-bit word. The output of the cipher function is the concatenation of these eight 4-bit words, permuted by a function named \(P\).

To compute the sixteen subkeys \(K_n\) (with \(n \in \{1, \ldots, 16\}\)), a function named \(PC1\) (permuted choice) selects 56 bits from the 64-bit key, and then splits the result into two 28-bit words, named \(C_0\) and \(D_0\). For iteration \(n\), the subkey \(K_n\) is obtained by application of a function named \(PC2\) to select 48 bits from \(C_nD_n\), which are defined by successive shift operations according to a schedule defined in the standard. Decryption follows the same scheme, only applying the subkeys in the opposite order.

3 Formal Models of the Asynchronous Data Encryption Standard

Figure 1 shows the overall architecture of the LOTOS and LNT models, which closely follow the architecture of the standard. Each block (permutation, S-box, bit-wise sum, shift, etc.) is represented by a process, communicating by rendezvous with its neighbors. Thus, there is no need for a global clock: each block waits for its operands, performs its operation, and transmits the result to the subsequent block.

In addition to the processes with a direct correspondence to blocks of the DES standard, the models include ten processes without a direct correspondence, but required to share blocks (or processes) among

\(^3\)Longer data must be split into 64-bit words, possibly adding zeros at the end so that the total number of bits is a multiple of 64. Each 64-bit word is then encrypted or decrypted separately.
Figure 1: Architecture of the asynchronous Data Encryption Standard. Boxes represent processes, and arrows indicate synchronizations (dashed arrows correspond to the control signals). Arrows are labeled by the name of the corresponding gate, where $F_K$ stands for $FIRST_K$, and $I_K$ stands for $INTERMEDIATE_K$. 
all iterations of the algorithm: the six processes of the CONTROLLER, and the four processes CHOOSE_L, CHOOSE_R, CHOOSE_K, and DUP_K. The latter four are arbiters and/or multiplexers, which select among several inputs and possibly duplicate their output.

The process COUNTER generates the control signal “CS !i”, where i starts from 0 and is increased up to 16, thus indicating seventeen steps (one more than the number of iterations in the algorithm). The signal CS is sent to the five other processes of the CONTROLLER, which in turn indicate to the shift register the number and direction of the shift(s) and to the four arbiters which input to read from and/or which output to write to. Seventeen steps are required because the processes CHOOSE_L and CHOOSE_R take as input either the initial data or the result of the previous iteration, and output either to the next iteration or the final output. Hence it is necessary to execute them before and after the sixteen iterations, explaining the additional step. Precisely, the rendezvous “CS !16” triggers a rendezvous only in process CTRL_MUX_LR, whereas it is simply consumed by CTRL_MUX_K, CTRL_DMUX_K, and CTRL_SHIFT.

Notice that the formal models contain no non-deterministic choice (select operator of LNT or “[]” operator of LOTOS), because every block of the DES is deterministic.

The complete LOTOS and LNT models are given as appendices and also available on the CADP website. Table 1 gives the number of lines for the two models, plus those of the LOTOS model generated by the LNT2LOTOS translator from the LNT model. The LNT model is significantly shorter and syntactically closer to the DES standard. In particular, the definition of the S-boxes using tables (as in the standard) is more convenient, due to the automatically defined functions to manipulate arrays. Compared to the hand-written models, the generated LOTOS model is much larger due to many automatically generated functions and auxiliary processes that are “inlined” in the hand-written models.

Rewriting the LOTOS model into LNT uncovered a few errors in the LOTOS definitions of the S-boxes and a small bug in the LNT2LOTOS translator. This rewrite also showed that the controller was too restrictive. Precisely, the five control processes (CTRL_MUX_K, CTRL_SHIFT, CTRL_DMUX_K, and the two instances of CTRL_MUX_LR) were synchronized at the end of an encoding or decoding, although CTRL_SHIFT could accept a new input on gate CRYPT as soon as the shift-command for the generation of the last subkey has been sent to the shift register.

4 Analysis of the Abstract Model

Analyzing the DES with enumerative techniques is challenging, because it is unfeasible to enumerate all 64-bit vectors. A first approach is to abstract from the actual data values and to focus the analysis on the control part, e.g., check whether the sixteen iterations are correctly synchronized. Redefining the BIT data type to contain a single value (instead of two values), automatically transforms all bit-vector types into singleton types and all functions operating on bit vectors to the identity function. Due to this drastic abstraction, enumerating all possible inputs is trivial, enabling LTS generation without resorting to environments and compositional techniques [7].

\[\text{http://cadp.inria.fr/demos/demo_38}\]

\[\text{See entry \#2076 in the list of changes to CADP (http://cadp.inria.fr/changes.html).}\]

\[\text{Removing this unnecessary synchronization (also present in the CHP and IF models) slightly increases the LTS size: the initial LOTOS model yielded an LTS with 588,785,433 states and 5,512,418,012 transitions.}\]

\[\text{The IF model [1] used a similar abstraction.}\]

\[\text{The significantly smaller size (5.3 million states and 30 million transitions) of the LTS corresponding to the IF model [1] is explained by the fact that in the IF model the eight S-boxes execute sequentially rather than in parallel [16, page 305, footnote 4]. Applying a similar restriction to the LNT model, the size of the corresponding LTS drops to 1,375,048 states and 7,804,352 transitions (respectively to 8,183,770 states and 45,025,227 transitions for a restricted LOTOS model). A further difference is}\]
Table 1: Line number count of the models

<table>
<thead>
<tr>
<th></th>
<th>LOTOS</th>
<th>LNT</th>
<th>gen. LOTOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>types &amp; functions</td>
<td>1712</td>
<td>375</td>
<td>2514</td>
</tr>
<tr>
<td>channels</td>
<td>0</td>
<td>50</td>
<td>58</td>
</tr>
<tr>
<td>processes</td>
<td>671</td>
<td>668</td>
<td>772</td>
</tr>
<tr>
<td>total</td>
<td>1843</td>
<td>1293</td>
<td>3344</td>
</tr>
</tbody>
</table>

Table 2: Direct LTS Generation

(on an Xeon(R) E5-2630 at 2.4 GHz)

<table>
<thead>
<tr>
<th></th>
<th>LOTOS</th>
<th>LNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>states</td>
<td>391,914,192</td>
<td>167,300,852</td>
</tr>
<tr>
<td>transitions</td>
<td>5,542,917,498</td>
<td>1,500,073,686</td>
</tr>
<tr>
<td>time (min)</td>
<td>228</td>
<td>66</td>
</tr>
<tr>
<td>RAM (GB)</td>
<td>19.13</td>
<td>4.93</td>
</tr>
</tbody>
</table>

Direct LTS Generation and Equivalence between the LNT and LOTOS Models. Table 2 gives statistics about the LTSs generated directly from the abstract models. The two models (LOTOS and LNT) yield LTSs equivalent for branching (but not strong) bisimulation (checked by PROPERTY 7 of the SVL script given in Appendix D). In both cases, the LTS minimized for branching bisimulation has 28 states and 78 transitions.

The significant difference in LTS size between the LNT model and the LOTOS model is explained by the semantic difference in the symmetric sequential composition [5]. The LOTOS operator “>>” generates an internal “i” transition, whereas the LNT operator “;” does not. Such a symmetric sequential composition (rather than a simple action prefix) is required whenever a process can read several inputs in parallel before producing its output. This is in particular the case for process P, which reads the output of the eight S-boxes. When these internal “i” transitions are removed by adding the pragma “(*! atomic *)” to all occurrences of “>>”, the LOTOS model yields an LTS of exactly the same size as the LNT model.

The LTS obtained after removing all offers (using appropriate renaming operations, i.e., those applied to file des_sample.bcg in PROPERTY 6 of the SVL script given in Appendix D) and minimization using branching bisimulation is shown in Figure 2.

Compositional LTS Generation. Compositional LTS generation, i.e., the bottom-up LTS construction alternating generation and minimization steps, is much more efficient. The initial steps of the SVL script given in Appendix D require at most 50 MB of RAM and generate the minimized LTS (28 states and 78 transitions) in less than 90 seconds. The success of compositional techniques on the LOTOS model triggered the development of the CHP2LOTOS translator [10] to provide the full power of CADP to the designers of asynchronous circuits.

Model Checking. Several properties of the control part have been analyzed formally, using model checking and equivalence checking on the LTS generated compositionally from the abstract models.

A first property, called PROPERTY 1 in the SVL script, expresses the absence of deadlocks.

A second property, called PROPERTY 2 expresses the fact that a triplet of inputs on gates CRYPT, DATA, and KEY is eventually followed by an rendezvous on gate OUTPUT.

A third property, called PROPERTY 3, describes the asynchronism of the DES models, i.e., that the DES may accept the inputs for N future rounds before it produces the result of the current round. This property is expressed by two temporal logic formulae, a first one expressing that the DES never accepts more than N inputs in advance, and a second one expressing that there exists an execution, where the DES indeed accepts N inputs in advance. This third property is parametric, because N varies for the three inputs DATA, CRYPT, and KEY.

A last property, called PROPERTY 4, expresses the correct synchronization between the data path and the key path, namely that each encoding or decoding executes the sixteen iterations. This property can that the controller in the IF model is a single automaton and enforces a stronger ordering between the inputs and outputs than the more asynchronous architecture shown in Figure 1.
be verified in two ways: by model checking a temporal logic formula and by checking the equivalence of the generated labeled transition system with a simple automaton containing a loop of sixteen SUBKEY transitions (because there must be a subkey per iteration) interleaved with a CRYPT transition (because each encoding or decoding requires one such transition). The LNT (respectively, LOTOS) model of this automaton is given in Appendix A.13 (respectively, B.14). Notice that this verification requires to keep the SUBKEY gate visible.

5 Analysis of the Concrete Model

The concrete models (i.e., without data abstraction) can be used to check the correctness of the results computed by the DES. Two different approaches were used: model checking and the generation of a prototype implementation.

**Rapid Prototyping.** Using the EXEC/CAESAR framework [11] for rapid prototyping, it is possible to generate a C implementation of the LOTOS or LNT model: one only has to provide C functions implementing the interaction with the environment on the visible gates CRYPT, DATA, KEY, and OUTPUT. This C code is given in Appendix C.

We first checked that the prototype is reversible, i.e., that deciphering a cipher with the same key results in the original data. These tests helped to spot and correct some errors in the subkey generation.

We also compared the results of the prototypes (LOTOS and LNT), and to those of some other publicly available implementations\(^9\). These comparisons helped to spot and correct a handful of differences

\(^9\)See for instance [https://www.schneier.com/books/applied_cryptography/source.html](https://www.schneier.com/books/applied_cryptography/source.html)
caused by typographic errors, i.e., bad copying of the DES standard into LOTOS and LNT. The compilation and execution of the prototype implementation corresponds to PROPERTY_5 in the SVL script given in Appendix D. Although using a handful of tests is far from exhaustive, the structure of the DES with its iterations, permutations and bit operations and the fact that cipher heavily shuffles the input data lead to a coverage sufficient for the purpose of analyzing the control part.

**Model Checking.** The generation of an LTS from the concrete model requires an environment to restrict the domain of possible input values. The model DES_SAMPLE is such a variant of the concrete one, including a sequential environment providing a key and a data for a single encryption and checking the correctness of the output. The direct generation of the corresponding LTS (10,156,715 states and 75,933,635 transitions for the LNT model, and 32,219,740 states and 259,010,596 transitions for the LOTOS model) requires less than 6 GB of RAM.

The correctness of the computed result can verified by model checking a property expressing that the action on gate OUTPUT is eventually reached. A second verification is that after removing all offers, the LTS is included in the LTS of Figure 2. These verifications are carried out by the PROPERTY_6 in the SVL script given in Appendix D.

### 6 Conclusion

This paper presents two formal models of the Data Encryption Standard, which might be an interesting benchmark example, because it is both complex and tractable (with current hardware and/or compositional techniques). For instance, it is sufficiently large to make interesting screenshots of the monitor of the distributed state space generation tool DISTRIBUTOR [9] (see also the DISTRIBUTOR manual page10). These models also illustrate an interesting feature of LOTOS and LNT, namely the possibility to easily change the implementation of a data type to transform a prototype implementation into an abstract model adapted to formal verification.

**Acknowledgements.** Some of the experiments presented in this paper were carried out using the Grid’5000 experimental testbed11 built by Inria with support from CNRS, RENATER, several Universities, and other funding bodies. I am grateful to Hubert Garavel for his suggestion to improve the model(s) and their presentation. I would like to thank Edith Beignè, François Bertrand, Pascal Vivet (CEA Leti), Dominique Borrione, Menouer Boubekeur, Marc Renaudin (TIMA), and Gwen Salaün (Inria) for discussions about asynchronous logic, the CHP language, and the implementation of the DES in asynchronous logic.

**References**


Asynchronous Data Encryption Standard


A LNT Model of an Asynchronous DES

This appendix gives the complete LNT model of the DES, as required by the SVL verification scenario given in Appendix D. This model requires CADP version 2015-h “Stony Brook” (August 2015) or later. The syntax and semantics of LNT is defined in the reference manual [3].

A.1 Module BIT_CONCRETE

module BIT_CONCRETE is

  type BIT is
    — bit data type with two different values
    0 !implemented by "BIT_ZERO",
    1 !implemented by "BIT_ONE"
    with "=="
  end type

end module

A.2 Module BIT_ABSTRACT

This module defines the abstraction replacing the concrete implementation of type BIT (see module BIT_CONCRETE.lnt above), transforming the two-valued type into a singleton, which transitively transforms all bit vectors into singletons as well, thus completely abstracting data.

module BIT_ABSTRACT is

  type BIT is
    — bit data type abstracted to a single value
    0
    with "=="
  end type

function 1 : BIT is
  return 0
end function
end module

A.3 Module TYPES

module TYPES (BIT) with "get" is

--- exclusive-or operation on (abstract or concrete) bits

function xor_ (X, Y : BIT) : BIT is
  if X = Y then
    return 0
  else
    return 1
  end if
end function

--- data type for counting the iterations of the algorithm

type ITERATION is
  range 0 .. 16 of NAT
  with "=", "|="
end type

--- data type for the controlling the behavior of the shift register

type SHIFT is
  NO, -- no shift
  LS1, -- 1 left shift
  LS2, -- 2 left shifts
  RS1, -- 1 right shift
  RS2 -- 2 right shifts
  with "="
end type

--- data type for the control signals used for the multiplexers

type PHASE is
  F, -- first iteration
  N, -- intermediate iteration
  L -- last iteration
  with "="
end type

--- 2-bit vectors
type BIT2 is
  MK_2 (B1, B2: BIT)
end type

-- conversion of a 2-bit vector into a natural number
-- note: the most significant bit is B1

function BIT2_TO_NAT (X: BIT2) : NAT is
  var N: NAT in
  N := 0;
  if X.B1 = 1 then N := N + 2 end if;
  if X.B2 = 1 then N := N + 1 end if;
  return N
  end var
end function

-- 4-bit vectors

type BIT4 is "implemented by" "ADT_BIT4"
  MK_4 (B1, B2, B3, B4: BIT) "implemented by" "MK_4"
end type

-- conversion of a 4-bit vector into a natural number
-- note: the most significant bit is B1

function BIT4_TO_NAT (X: BIT4) : NAT is
  var N: NAT in
  N := 0;
  if X.B1 = 1 then N := N + 8 end if;
  if X.B2 = 1 then N := N + 4 end if;
  if X.B3 = 1 then N := N + 2 end if;
  if X.B4 = 1 then N := N + 1 end if;
  return N
  end var
end function

-- conversion of a natural number into a 4-bit vector
-- note: the most significant bit is B1

function NAT_TO_BIT4 (N: NAT) : BIT4 is
  case N in
  0 -> return MK_4 (0, 0, 0, 0)
  | 1 -> return MK_4 (0, 0, 0, 1)
  | 2 -> return MK_4 (0, 0, 1, 0)
  | 3 -> return MK_4 (0, 0, 1, 1)
  | 4 -> return MK_4 (0, 1, 0, 0)
  | 5 -> return MK_4 (0, 1, 0, 1)
  | 6 -> return MK_4 (0, 1, 1, 0)
Asynchronous Data Encryption Standard

| 7 | return MK_4 (0, 1, 1, 1) |
| 8 | return MK_4 (1, 0, 0, 0) |
| 9 | return MK_4 (1, 0, 0, 1) |
|10 | return MK_4 (1, 0, 1, 0) |
|11 | return MK_4 (1, 0, 1, 1) |
|12 | return MK_4 (1, 1, 0, 0) |
|13 | return MK_4 (1, 1, 0, 1) |
|14 | return MK_4 (1, 1, 1, 0) |
| any | return MK_4 (1, 1, 1, 1) |

end case
end function

--- 6-bit vectors

type BIT6 is
  MK_6 (B1, B2, B3, B4, B5, B6: BIT)
end type

--- projection of 6-bit vectors to 2-bit vectors

function 1AND6 (X: BIT6) : BIT2 is
  return MK_2 (X.B1, X.B6)
end function

--- projection of 6-bit vectors to 4-bit vectors

function 2TO5 (X: BIT6) : BIT4 is
  return MK_4 (X.B2, X.B3, X.B4, X.B5)
end function

--- 32-bit vectors

type BIT32 is
  MK_32 (B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12, B13, B14, B15, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B29, B30, B31, B32: BIT)
end type

--- bitwise XOR for 32-bit vectors

function XOR (A, B: BIT32) : BIT32 is
  return MK_32 (A.B1 xor B.B1, A.B2 xor B.B2, A.B3 xor B.B3, A.B4 xor B.B4, A.B5 xor B.B5, A.B6 xor B.B6, A.B7 xor B.B7, A.B8 xor B.B8, A.B9 xor B.B9, A.B10 xor B.B10,
W. Serwe

A. B11 xor B. B11, A. B12 xor B. B12,
A. B13 xor B. B13, A. B14 xor B. B14,
A. B15 xor B. B15, A. B16 xor B. B16,
A. B17 xor B. B17, A. B18 xor B. B18,
A. B19 xor B. B19, A. B20 xor B. B20,
A. B21 xor B. B21, A. B22 xor B. B22,
A. B23 xor B. B23, A. B24 xor B. B24,
A. B25 xor B. B25, A. B26 xor B. B26,
A. B27 xor B. B27, A. B28 xor B. B28,
A. B29 xor B. B29, A. B30 xor B. B30,
A. B31 xor B. B31, A. B32 xor B. B32)

end function

— concatenation of eight 4-bit vectors to form a 32-bit vector

function MK_32 (V1, V2, V3, V4, V5, V6, V7, V8: BIT4) : BIT32 is
end function

— 48-bit vectors

type BIT48 is
  MK_48 (B1, B2, B3, B4, B5, B6, B7, B8,
          B9, B10, B11, B12, B13, B14, B15, B16,
          B17, B18, B19, B20, B21, B22, B23, B24,
          B25, B26, B27, B28, B29, B30, B31, B32,
          B33, B34, B35, B36, B37, B38, B39, B40,
          B41, B42, B43, B44, B45, B46, B47, B48: BIT)
end type

— bitwise XOR for 48-bit vectors

function XOR (A, B: BIT48) : BIT48 is
  return MK_48 (A. B1 xor B. B1, A. B2 xor B. B2,
                 A. B3 xor B. B3, A. B4 xor B. B4,
                 A. B5 xor B. B5, A. B6 xor B. B6,
                 A. B7 xor B. B7, A. B8 xor B. B8,
                 A. B9 xor B. B9, A. B10 xor B. B10,
                 A. B11 xor B. B11, A. B12 xor B. B12,
                 A. B13 xor B. B13, A. B14 xor B. B14,
                 A. B15 xor B. B15, A. B16 xor B. B16,
                 A. B17 xor B. B17, A. B18 xor B. B18,
                 A. B19 xor B. B19, A. B20 xor B. B20,
                 A. B21 xor B. B21, A. B22 xor B. B22,
                 A. B23 xor B. B23, A. B24 xor B. B24,
                 A. B25 xor B. B25, A. B26 xor B. B26,
Asynchronous Data Encryption Standard

\[
\begin{align*}
A \cdot B_{27} & \oplus B \cdot B_{27}, \quad A \cdot B_{28} \oplus B \cdot B_{28}, \\
A \cdot B_{29} & \oplus B \cdot B_{29}, \quad A \cdot B_{30} \oplus B \cdot B_{30}, \\
A \cdot B_{31} & \oplus B \cdot B_{31}, \quad A \cdot B_{32} \oplus B \cdot B_{32}, \\
A \cdot B_{33} & \oplus B \cdot B_{33}, \quad A \cdot B_{34} \oplus B \cdot B_{34}, \\
A \cdot B_{35} & \oplus B \cdot B_{35}, \quad A \cdot B_{36} \oplus B \cdot B_{36}, \\
A \cdot B_{37} & \oplus B \cdot B_{37}, \quad A \cdot B_{38} \oplus B \cdot B_{38}, \\
A \cdot B_{39} & \oplus B \cdot B_{39}, \quad A \cdot B_{40} \oplus B \cdot B_{40}, \\
A \cdot B_{41} & \oplus B \cdot B_{41}, \quad A \cdot B_{42} \oplus B \cdot B_{42}, \\
A \cdot B_{43} & \oplus B \cdot B_{43}, \quad A \cdot B_{44} \oplus B \cdot B_{44}, \\
A \cdot B_{45} & \oplus B \cdot B_{45}, \quad A \cdot B_{46} \oplus B \cdot B_{46}, \\
A \cdot B_{47} & \oplus B \cdot B_{47}, \quad A \cdot B_{48} \oplus B \cdot B_{48}
\end{align*}
\]

end function

| projections of 48-bit vectors to 6-bit vectors |

function 1 TO 6 (X: BIT48) : BIT6 is
end function

function 7 TO 12 (X: BIT48) : BIT6 is
end function

function 13 TO 18 (X: BIT48) : BIT6 is
end function

function 19 TO 24 (X: BIT48) : BIT6 is
end function

function 25 TO 30 (X: BIT48) : BIT6 is
end function

function 31 TO 36 (X: BIT48) : BIT6 is
end function

function 37 TO 42 (X: BIT48) : BIT6 is
end function

function 43 TO 48 (X: BIT48) : BIT6 is
end function

| 56-bit vectors |

type BIT56 is
MK_56 (B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12, B13, B14, B15, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B29, B30, B31, B32, B33, B34, B35, B36, B37, B38, B39, B40, B41, B42, B43, B44, B45, B46, B47, B48, B49, B50, B51, B52, B53, B54, B55, B56: BIT)

end type

— left shift of a 56-bit vector
— note: more precisely, parallel left shift of two 28-bit vectors

function LSHIFT (X: BIT56) : BIT56 is
end function

— right shift of a 56-bit vector
— note: more precisely, parallel right shift of two 28-bit vectors

function RSHIFT (X: BIT56) : BIT56 is
end function

— 64-bit vectors

type BIT64 is !implementedby "ADT_BIT64" !printedby "ADT_PRINT_BIT64"
  MK_64 (B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12, B13, B14, B15, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B29, B30, B31, B32, B33, B34, B35, B36, B37, B38, B39, B40, B41, B42, B43, B44, B45, B46, B47, B48, B49, B50, B51, B52, B53, B54, B55, B56, B57, B58, B59, B60, B61, B62, B63, B64: BIT)
!implementedby "MK_64"
  with "=" , " /= "
end type
Asynchronous Data Encryption Standard

— projections of 64-bit vectors to 32-bit vectors

function 1TO32 (X : BIT64) : BIT32 is
return MK_32 (X.B1, X.B2, X.B3, X.B4, X.B5, X.B6, X.B7, X.B8,
end function

function 33TO64 (X : BIT64) : BIT32 is
end function

— concatenation of sixteen 4-bit vectors to form a 64-bit vector

function MK_64 (V1, V2, V3, V4, V5, V6, V7, V8,
V9, V10, V11, V12, V13, V14, V15, V16 : BIT4) : BIT64 is
!implemented by "CONCAT_BIT4"
end function

— concatenation of two 32-bit vectors to form a 64-bit vector

function MK_64 (V1, V2 : BIT32) : BIT64 is
A.4 Module PERMUTATION_FUNCTIONS

module PERMUTATION_FUNCTIONS (TYPES) is

--- E: expansion of a 32-bit vector to a 48-bit vector

function E (X : BIT32) : BIT48 is
end function

--- IP: initial permutation

function IP (X : BIT64) : BIT64 is
end function

--- IIP: inverse initial permutation

function IIP (X : BIT64) : BIT64 is
Asynchronous Data Encryption Standard


end function

--- P: permutation

function P (X : BIT32) : BIT32 is
  return MK_32 (X. B16, X. B7, X. B20, X. B21,
  X. B29, X. B12, X. B28, X. B17,
  X. B1, X. B15, X. B23, X. B26,
  X. B5, X. B18, X. B31, X. B10,
  X. B2, X. B8, X. B24, X. B14,
  X. B32, X. B27, X. B3, X. B9,
  X. B19, X. B13, X. B30, X. B6,
end function

--- PC1: permuted choice 1

function PC1 (X : BIT64) : BIT56 is
  return MK_56 (X. B57, X. B49, X. B41, X. B33, X. B25, X. B17, X. B9,
  X. B1, X. B58, X. B50, X. B42, X. B34, X. B26, X. B18,
  X. B14, X. B6, X. B61, X. B53, X. B45, X. B37, X. B29,
end function

--- PC2: permuted choice 2

function PC2 (X : BIT56) : BIT48 is
  return MK_48 (X. B14, X. B17, X. B11, X. B24, X. B1, X. B5,
  X. B3, X. B28, X. B15, X. B6, X. B21, X. B10,
  X. B23, X. B19, X. B12, X. B4, X. B26, X. B8,
  X. B41, X. B52, X. B31, X. B37, X. B47, X. B55,
  X. B30, X. B40, X. B51, X. B45, X. B33, X. B48,
  X. B44, X. B49, X. B39, X. B56, X. B34, X. B53,
end function

end module
A.5 Module S_BOX_FUNCTIONS

The LNT model directly encodes the S-box tables as two-dimensional arrays, requiring the definition of the additional data types ROW and _S_BOX_ARRAY_, together with accessor functions `GET_ROW()` and `GET_COLUMN()` and projection functions `1AND6()` and `2TO5()` (the latter are part of module TYPES (see Appendix A.3).

```plaintext
module S_BOX_FUNCTIONS (TYPES) is

-- data types defining a two-dimensional array to encode the S-boxes

type ROW is
  array [0..15] of NAT
end type

type S_BOX_ARRAY is
  array [0..3] of ROW
end type

function GET_ROW (X : BIT6) : NAT is
  return BIT2_TO_NAT (1AND6 (X))
end function

function GET_COLUMN (X : BIT6) : NAT is
  return BIT4_TO_NAT (2TO5 (X))
end function

function S1 : S_BOX_ARRAY is
  return S_BOX_ARRAY
    (ROW (14,  4, 13,  1,  2, 15, 11,  8,  3, 10,  6, 12,  5,  9,  0,  7),
     ROW ( 0, 15,  7,  4, 14,  2, 13,  1, 10,  6, 12, 11,  9,  5,  3,  8),
     ROW ( 4,  1, 14,  8, 13,  6,  2, 11, 15, 12,  9,  7,  3, 10,  5,  0),
     ROW (15, 12,  8,  2,  4,  9,  1,  7,  5, 11,  3, 14, 10,  0,  6, 13))
end function

function S2 : S_BOX_ARRAY is
  return S_BOX_ARRAY
    (ROW (15,  1,  8, 14,  6, 11,  3,  4,  9,  7,  2, 13, 12,  0,  5, 10),
     ROW ( 3, 13,  4,  7, 15,  2,  8, 14, 12,  0,  1, 10,  6,  9, 11,  5),
     ROW ( 0, 14,  7, 11, 10,  4, 13,  1,  5,  8, 12,  6,  9,  3,  2, 15),
     ROW (13,  8, 10,  1,  3, 15,  4,  2, 11,  6,  7, 12,  0,  5, 14,  9))
end function
```
function S3 : S_BOX_ARRAY is  
  return S_BOX_ARRAY  
    (ROW (10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8),  
     ROW (13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1),  
     ROW (13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7),  
     ROW (1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12))  
end function

function S4 : S_BOX_ARRAY is  
  return S_BOX_ARRAY  
    (ROW (7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15),  
     ROW (13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9),  
     ROW (10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4),  
     ROW (3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14))  
end function

function S5 : S_BOX_ARRAY is  
  return S_BOX_ARRAY  
    (ROW (2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9),  
     ROW (14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6),  
     ROW (4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14),  
     ROW (11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3))  
end function

function S6 : S_BOX_ARRAY is  
  return S_BOX_ARRAY  
    (ROW (12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11),  
     ROW (10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8),  
     ROW (9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6),  
     ROW (4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13))  
end function

function S7 : S_BOX_ARRAY is  
  return S_BOX_ARRAY  
    (ROW (4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1),  
     ROW (13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6),  
     ROW (1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2),  
     ROW (6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12))  
end function
function S8 : S_BOX_ARRAY is
return S_BOX_ARRAY
   (ROW (13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7),
    ROW (1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2),
    ROW (7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8),
    ROW (2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11))
end function

end module

A.6 Module CHANNELS

module CHANNELS (TYPES) is

-- channel types for bit vectors

channel C4 is
   (BIT4)
end channel

channel C6 is
   (BIT6)
end channel

channel C32 is
   (BIT32)
end channel

channel C48 is
   (BIT48)
end channel

channel C56 is
   (BIT56)
end channel

channel C64 is
   (BIT64)
end channel

-- channel types for scalar data types

channel CB is
   (BOOL)
end channel

channel CIT is
   (ITERATION)
end channel
channel CP is
  (PHASE)
end channel

channel CS is
  (SHIFT)
end channel

end module

A.7 Module CONTROLLER

module CONTROLLER (CHANNELS) is

-contained
CONTROLLER executes five concurrent processes controlling the various
- multiplexers and the shift register, which are synchronized via a sixth
- process counting the iterations

process CONTROLLER [CRYPT: CB,
  CTRL_CL, CTRL_CR: CP,
  CTRL_SHIFT: CS, CTRL_DK, CTRL_CK: CP] is

  hide CS: CIT in
  par CS in
    COUNTER [CS]
    ||
    CTRL_MUX_LR [CS, CTRL_CL]
    ||
    CTRL_MUX_LR [CS, CTRL_CR]
    ||
    CTRL_SHIFT [CRYPT, CS, CTRL_SHIFT]
    ||
    CTRL_DMUX_K [CS, CTRL_DK]
    ||
    CTRL_MUX_K [CS, CTRL_CK]
  end par
end hide
end process

-contained
COUNTER counts the iterations (modulo 17)

process COUNTER [CS: CIT] is

  var IT: ITERATION in
  IT := 0;
  loop
    CS (IT);
    IT := ITERATION (((NAT (IT) + 1) mod 17))
  end loop
end var
end process
CTRL_MUX_LR generates 1 F, then 15 Ns, and finally 1 L to control the multiplexers in the data path

process CTRL_MUX_LR [CS: CIT, CTRL: CP] is
  var IT: ITERATION in
  loop
    CS (?IT);
    if IT == (0 of ITERATION) then
      CTRL (F)
    elsif IT == last then
      CTRL (L)
    else
      CTRL (N)
    end if
  end loop
end var
end process

CTRL_SHIFT controls the shift register

process CTRL_SHIFT [CRYPT: CB, CS: CIT, CTRL: CS] is
  var CRYPT: BOOL, IT: ITERATION in
  loop
    CRYPT (?CRYPT);
    loop LL in
      CS (?IT);
      if IT == last then
        break LL
      elsif CRYPT then
        if (IT == (0 of ITERATION)) or (IT == (1 of ITERATION)) or
           (IT == (8 of ITERATION)) or (IT == (15 of ITERATION))
          then
            CTRL (LS1)
          else
            CTRL (LS2)
        end if
      else
        if IT == (0 of ITERATION) then
          CTRL (NO)
        elsif (IT == (1 of ITERATION)) or
           (IT == (8 of ITERATION)) or
           (IT == (15 of ITERATION))
          then
            CTRL (RS1)
          else
            CTRL (RS2)
        end if
      end if
    end loop
  end loop
-- CTRL_DMUX_K generates 15 Ns and 1 L to control the doubling multiplexer

process CTRL_DMUX_K [CS: CIT, CTRL: CP] is
var IT: ITERATION in
loop
CS (?IT);
if IT == (15 of ITERATION) then
CTRL (L)
elif IT != last then
CTRL (N)
end if
end loop
end var
end process

-- CTRL_MUX_K generates 1 F and 15 Ns to control the key path multiplexer

process CTRL_MUX_K [CS: CIT, CTRL: CP] is
var IT: ITERATION in
loop
CS (?IT);
if IT == (0 of ITERATION) then
CTRL (F)
elif IT != last then
CTRL (N)
end if
end loop
end var
end process

end module

A.8 Module KEY_PATH

module KEY_PATH (CHANNELS) is

-- KEY_PATH generates the 16 subkeys of the key schedule

process KEY_PATH [KEY: C64, SUBKEY: C48,
CTRL_SHIFT: CS, CTRL_DK, CTRL_CK: CP] is
hide FIRST_K, INTERMEDIATE_K: C56 in
par
FIRST_K ->
PC1 [KEY, FIRST_K]
FIRST_K, INTERMEDIATE_K ->
  hide K, KKK, SK: C56 in
  par
    K, SK ->
      SHIFT_REGISTER [CTRL_SHIFT, K, SK]
    ||
    SK, KKK ->
      DUPLICATE_K [CTRL_DK, SK, INTERMEDIATE_K, KKK]
    ||
    KKK, K ->
      CHOOSE_K [CTRL_CK, FIRST_K, KKK, K]
  end par
end hide
||
INTERMEDIATE_K ->
  PC2 [INTERMEDIATE_K, SUBKEY]
end par
end hide
end process

— PCI applies PCI to select a 56-bit vector from the initial 64-bit key

process PCI [KEY: C64, FIRST_K: C56] is
  var K64: BIT64 in
  loop
    KEY (?K64);
    FIRST_K (PCI (K64))
  end loop
end var
end process

— SHIFT_REGISTER performs, depending on the iteration, one or two shifts
— to the left or right of a 56-bit word

process SHIFT_REGISTER [CTRL: CS, INPUT, OUTPUT: C56] is
  var CTRL: SHIFT, I56: BIT56 in
  loop
    par
      CTRL (?CTRL)
    ||
      INPUT (?I56)
  end par;
  case CTRL in
    NO ->
      OUTPUT (I56)
    | LS1 ->
      OUTPUT (LSHIFT (I56))
    | LS2 ->
      OUTPUT (LSHIFT (LSHIFT (I56)))
    | RS1 ->
      OUTPUT (RSHIFT (I56))
    | RS2 ->
      OUTPUT (RSHIFT (RSHIFT (I56)))
  end case
end loop
Asynchronous Data Encryption Standard

```vhdl
OUTPUT (RSHIFT (I56))
| RS2 ->
OUTPUT (RSHIFT (RSHIFT (I56)))
end case
end loop
end var
end process

---
DUPLICATE_K reads a 56-bit vector from INPUT and outputs it to OUTPUT1 and OUTPUT2, but for the last iteration, where it outputs only to OUTPUT1. Because DUPLICATE_K always reads from INPUT, the order of the rendezvous on CTRL and INPUT is arbitrary.

process DUPLICATE_K [CTRL: CP, INPUT, OUTPUT1, OUTPUT2: C56] is
  var CTRL: PHASE, I56:BIT56 in
  loop
    par
      CTRL (?CTRL)
      |
      INPUT (?I56)
    end par;
    if CTRL = L then
      OUTPUT1 (I56)
    else — assert (CTRL = N)
      par
        |
        OUTPUT1 (I56)
        |
        OUTPUT2 (I56)
      end par
    end if
  end loop
end var
end process

---
CHOOSE_K closes the loop in the key path, by redirecting the input on INPUT to OUTPUT, but for the first iteration where the original key is read on FIRST_IN

process CHOOSE_K [CTRL: CP, FIRST_IN, INPUT, OUTPUT: C56] is
  var CTRL: PHASE, I56:BIT56 in
  loop
    CTRL (?CTRL);
    if CTRL = F then
      FIRST_IN (?I56)
    else — assert (CTRL = N)
      INPUT (?I56)
    end if:
    OUTPUT (I56)
  end loop
end var
```
end process

-- PC2 applies PC2 to generate the current subkey

process PC2 [KK: C56, SUBKEY: C48] is
  var I56: BIT56 in
  loop
    KK (I56);
    SUBKEY (PC2 (I56))
  end loop
end var
end process

end module

A.9 Module DATA_PATH

module DATA_PATH (PERMUTATION_FUNCTIONS, CIPHER) is

-- DATA_PATH performs the 16 iterations ciphering DATA with the subkeys
-- read on gate SUBKEY

process DATA_PATH [DATA, OUTPUT: C64, SUBKEY: C48, CTRL_CL, CTRL_CR: CP] is
  hide FIRST_L, FIRST_R, OUTPUT_L, OUTPUT_R: C32 in
  par
    FIRST_L, FIRST_R ->
      IP [DATA, FIRST_L, FIRST_R]
  ||
    FIRST_L, FIRST_R, OUTPUT_L, OUTPUT_R ->
    hide CL_XR, CR_FX, FX_XR, XR_CR, CR_CL: C32 in
    par
      CR_CL, CL_XR ->
        CHOOSE_L [CTRL_CL, FIRST_L, CR_CL, CL_XR, OUTPUT_L]
    ||
      XR_CR, CR_CL, CR_FX ->
    ||
      CR_FX, FX_XR ->
        CIPHER [SUBKEY, CR_FX, FX_XR]
    ||
      CL_XR, FX_XR, XR_CR ->
        XOR_32 [CL_XR, FX_XR, XR_CR]
    end par
  end hide
  ||
    OUTPUT_L, OUTPUT_R ->
    IIIP [OUTPUT_L, OUTPUT_R, OUTPUT]
  end par
end hide
end process
--- IP applies the initial permutation IP to the initial 64-bit vector received on gate DATA and breaks the resulting 64-bit vector into two 32-bit vectors L and R

```vhdl
process IP [DATA: C64, FIRST_L, FIRST_R: C32] is
  var I64: BIT64 in
  loop
    DATA (?I64);
    I64 := IP (I64);
  par
    FIRST_L (1TO32 (I64))
    FIRST_R (33TO64 (I64))
  end par
  end loop
end var
end process
```

--- CHOOSE_L reads a 32-bit vector from INPUT and outputs to OUTPUT, but for the first iteration, where it reads from FIRST_IN, and the last iteration, where it outputs to LAST_OUT

```vhdl
process CHOOSE_L [CTRL: CP, FIRST_IN, INPUT, OUTPUT, LAST_OUT: C32] is
  var CTRL: PHASE, L32: BIT32 in
  loop
    CTRL (?CTRL);
    case CTRL in
      F =>
        FIRST_IN (?L32);
        OUTPUT (L32)
      | N =>
        INPUT (?L32);
        OUTPUT (L32)
      | L =>
        INPUT (?L32);
        LAST_OUT (L32)
    end case
  end loop
end var
end process
```

--- CHOOSE_R reads a 32-bit vector from INPUT and outputs to OUT1 and OUT2, but for the first iteration, where it reads from FIRST_IN, and the last one, where it outputs to LAST_OUT

```vhdl
process CHOOSE_R [CTRL: CP, FIRST_IN, INPUT, OUT1, OUT2, LAST_OUT: C32] is
  var CTRL: PHASE, R32: BIT32 in
  loop
    case CTRL in
      F =>
        FIRST_IN (?R32);
        OUT1 (R32)
      | N =>
        INPUT (?R32);
        OUT1 (R32)
      | L =>
        INPUT (?R32);
        LAST_OUT (R32)
    end case
  end loop
end var
end process
```
CTRL (?CTRL);
case CTRL in
  F ->
    FIRST.IN (?R32);
    par
      OUT1 (R32)
    end par
  | N ->
    INPUT (?R32);
    par
      OUT1 (R32)
    end par
  | L ->
    INPUT (?R32);
    LAST.OUT (R32)
end case
end loop
end var
end process

--- XOR_32 asynchronously reads two 32-bit vectors and outputs their bitwise sum
---

process XOR_32 [A, B, R: C32] is
  var A32, B32: BIT32 in
  loop
    par
      A (?A32)
    end par;
    B (?B32)
    end loop
  end var
end process

--- IIIP assembles the two 32-bit vectors computed by the algorithm to a 64-bit vector and applies the inverse initial permutation IIIP to compute the final result
---

process IIIP [OUTPUT_L, OUTPUT_R: C32, OUTPUT: C64] is
  var OL, OH: BIT32 in
  loop
    par
      OUTPUT.L (?OL)
    end par
  end loop
end process
end process
end module

A.10 Module CIPHER

module CIPHER (CHANNELS, S_BOX_FUNCTIONS) is

-- processes implementing the cipher function according to Fig. 2 of [DES]

-- CIPHER computes \[ F(R, K) = P(S_i(E(R) \oplus K)) \]

process CIPHER [K: C48, R, PX: C32] is
  hide ER: C48,
  IS1, IS2, IS3, IS4, IS5, IS6, IS7, IS8: C6,
  SO1, SO2, SO3, SO4, SO5, SO6, SO7, SO8: C4
  in
    par
      ER ->
        E [R, ER]
      ||
        ER, IS1, IS2, IS3, IS4, IS5, IS6, IS7, IS8 ->
          XOR_48 [ER, K, IS1, IS2, IS3, IS4, IS5, IS6, IS7, IS8]
      ||
        IS1, IS2, IS3, IS4, IS5, IS6, IS7, IS8,
        SO1, SO2, SO3, SO4, SO5, SO6, SO7, SO8 ->
          par
            S1 [IS1, SO1]
            ||
            S2 [IS2, SO2]
            ||
            S3 [IS3, SO3]
            ||
            S4 [IS4, SO4]
            ||
            S5 [IS5, SO5]
            ||
            S6 [IS6, SO6]
            ||
            S7 [IS7, SO7]
            ||
            S8 [IS8, SO8]
          end par
      ||
        SO1, SO2, SO3, SO4, SO5, SO6, SO7, SO8 ->
        P [SO1, SO2, SO3, SO4, SO5, SO6, SO7, SO8, PX]
  end par;
process E [INPUT: C32, OUTPUT: C48] is
  var I32 : BIT32 in
  loop
    INPUT (?I32);
    OUTPUT (E(I32))
  end loop
  end var
end process

---

process XOR48 [A, B: C48, R1, R2, R3, R4, R5, R6, R7, R8: C6] is
  var A48, B48, I48 : BIT48 in
  loop
    par
      A (?A48)
      ||
      B (?B48)
    end par;
    I48 := XOR (A48, B48);
    par
      R1 (1TO6 (I48))
      ||
      R2 (7TO12 (I48))
      ||
      R3 (13TO18 (I48))
      ||
      R4 (19TO24 (I48))
      ||
      R5 (25TO30 (I48))
      ||
      R6 (31TO36 (I48))
      ||
      R7 (37TO42 (I48))
      ||
      R8 (43TO48 (I48))
    end par
  end loop
  end var
end process

---

E expands a 32-bit word to a 48-bit word using function E

XOR48 asynchronously reads two 48-bit vectors and outputs their bitwise sum, split into eight 6-bit vectors
process S1 [INPUT: C6, OUTPUT: C4] is
var I6 : BIT6 in
  loop
    INPUT (?I6);
    OUTPUT (NAT_TO_BIT4 (S1[GET_ROW (I6)][GET_COLUMN (I6)]))
  end loop
end var
end process

process S2 [INPUT: C6, OUTPUT: C4] is
var I6 : BIT6 in
  loop
    INPUT (?I6);
    OUTPUT (NAT_TO_BIT4 (S2[GET_ROW (I6)][GET_COLUMN (I6)]))
  end loop
end var
end process

process S3 [INPUT: C6, OUTPUT: C4] is
var I6 : BIT6 in
  loop
    INPUT (?I6);
    OUTPUT (NAT_TO_BIT4 (S3[GET_ROW (I6)][GET_COLUMN (I6)]))
  end loop
end var
end process

process S4 [INPUT: C6, OUTPUT: C4] is
var I6 : BIT6 in
  loop
    INPUT (?I6);
    OUTPUT (NAT_TO_BIT4 (S4[GET_ROW (I6)][GET_COLUMN (I6)]))
  end loop
end var
end process

process S5 [INPUT: C6, OUTPUT: C4] is
var I6 : BIT6 in
  loop
    INPUT (?I6);
    OUTPUT (NAT_TO_BIT4 (S5[GET_ROW (I6)][GET_COLUMN (I6)]))
  end loop
end var
end process
process S6 [INPUT: C6, OUTPUT: C4] is
  var I6 : BIT6 in
  loop
    INPUT (?I6);
    OUTPUT (NAT_TO_BIT4 (S6[GET_ROW (I6)][GET_COLUMN (I6)]))
  end loop
end var
end process

process S7 [INPUT: C6, OUTPUT: C4] is
  var I6 : BIT6 in
  loop
    INPUT (?I6);
    OUTPUT (NAT_TO_BIT4 (S7[GET_ROW (I6)][GET_COLUMN (I6)]))
  end loop
end var
end process

process S8 [INPUT: C6, OUTPUT: C4] is
  var I6 : BIT6 in
  loop
    INPUT (?I6);
    OUTPUT (NAT_TO_BIT4 (S8[GET_ROW (I6)][GET_COLUMN (I6)]))
  end loop
end var
end process

— P collects the results of the eight processes S BOX i (on INi) and
— outputs them in a single transition exit; the permutation P is applied
— in a second step when outputting the result on OUTPUT.

process P [IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8: C4, OUTPUT: C32] is
  var I1, I2, I3, I4, I5, I6, I7, I8 : BIT4 in
  loop
    par
      IN1 (?I1)
      | | IN2 (?I2)
      | | IN3 (?I3)
      | | IN4 (?I4)
      | | IN5 (?I5)
A.11 Module DES

The module DES.inl defines the architecture of the asynchronous DES together with the principal process MAIN instantiating process DES.

```vhdl
module DES (CONTROLLER, DATA_PATH, KEY_PATH) is

process DES [CRYPT : CB, KEY, DATA, OUTPUT: C64] is
hide SUBKEY: C48,
CTRL_CL, CTRL_CR: CP,
CTRL_SHIFT: CS, CTRL_DK, CTRL_CK: CP
in
par
  SUBKEY, CTRL_CL, CTRL_CR ->
  DATA_PATH [DATA, OUTPUT, SUBKEY, CTRL_CL, CTRL_CR]
||
  SUBKEY, CTRL_CK, CTRL_SHIFT, CTRL_DK ->
  KEY_PATH [KEY, SUBKEY, CTRL_SHIFT, CTRL_DK, CTRL_CK]
||
  CTRL_CL, CTRL_CR, CTRL_CK, CTRL_SHIFT, CTRL_DK ->
  CONTROLLER [CRYPT, CTRL_CL, CTRL_CR, CTRL_SHIFT, CTRL_DK, CTRL_CK]
end par
end hide
end process

process MAIN [CRYPT : CB, KEY, DATA, OUTPUT: C64] is
  DES [CRYPT, KEY, DATA, OUTPUT]
end process
end module
```

A.12 Model with Concrete Bits and Environment: DES_SAMPLE

This model instantiates the DES in a sequential environment providing input data and checking the output. Hence, this model can be used to directly (i.e., non compositionally) generate an LTS for the
concrete (and thus also the abstract) model. After hiding all offers and minimization for branching bisimulation, the generated LTS is the one shown in Figure 2.

```plaintext
module DES_SAMPLE (DES) is

-- DES_SAMPLE uses concrete bits

process MAIN_SAMPLE [CRYPT : CB, KEY, DATA, OUTPUT: C64] is
  par CRYPT, KEY, DATA, OUTPUT in
    DES [CRYPT, KEY, DATA, OUTPUT]
    ||
    ENVIRONMENT [CRYPT, KEY, DATA, OUTPUT]
  end par
end process

-- 64-bit constant frequently used as example for a key

function c.13345779_9BBCDFF1 : BIT64 is
  return MK_64 (0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1)
end function

-- 64-bit constant frequently used as example data

function C_01234567_89ABCDEF : BIT64 is
  return MK_64 (0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1)
end function

-- result of ciphering C_01234567_89ABCDEF with C_13345779_9BBCDFF1

function C_85E81354_0F0A405 : BIT64 is
  return MK_64 (1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1)
end function

-- process simulating the environment in order to close the system.
-- executing a single encryption of C_01234567_89abcdef with
-- c_13345779_9BBCDFF1

process ENVIRONMENT [CRYPT: CB, KEY, DATA, OUTPUT: C64] is
  CRYPT (true);
```
Asynchronous Data Encryption Standard

KEY (c_13345779_9BBCDFF1);
DATA (c_01234567_89abcdef);
— cipher of c_01234567_89abcdef with c_13345779_9BBCDFF1
OUTPUT (c_85e81354_0f0ab405);
stop
end process
end module

A.13 Module PROPERTY.4

module property.4 is

— synchronization channel without any offer

channel none is
 ()
end channel

— process describing a loop starting with a synchronization on gate CRYPT,
— followed by 16 synchronization on gate SUBKEY, where the synchronization
— on CRYPT corresponding to the next iteration can already appear after
— 14 synchronizations on SUBKEY.
— this process is used to verify the presence of 16 iterations in the DES.

process MAIN [CRYPT, SUBKEY: none] is
CRYPT;
loop
 var IT : NAT in
 for IT := 0 while IT < 14 by IT := IT + 1 loop
 SUBKEY
 end loop
end var;
par
 SUBKEY; SUBKEY
 |
 CRYPT
end par
end loop
end process
end module

B LOTOS Specification of an Asynchronous DES

This appendix gives the complete LOTOS models of the DES, as required by the SVL verification sce-

nario given in Appendix D. For an introduction to LOTOS, its syntax and semantics, see the international
standard [12] or one of the tutorials listed on the CADP website\textsuperscript{12}. In the sequel, a LOTOS model is called a “specification”, in conformance with the terminology of the LOTOS standard.

**B.1 Library BIT\_CONCRETE**

The definition of type BIT by library BIT\_CONCRETE is not strictly necessary, because one could use the library BIT provided with the CADP toolbox, which defines many more operations than just xor. Thus, the library BIT\_CONCRETE is included here for better comparison with its abstract version (see Appendix B.2 below).

\begin{verbatim}
type BIT is BOOLEAN
   sorts BIT
   opns 0 (∗! implementedby BIT\_ZERO constructor ∗) : → BIT
         1 (∗! implementedby BIT\_ONE constructor ∗) : → BIT
endtype
\end{verbatim}

**B.2 Library BIT\_ABSTRACT**

This library is a replacement of the type BIT, transforming the two-valued type into a singleton, which transitively transforms all bit vectors into singletons as well, thus completely abstracting data.

\begin{verbatim}
type BIT is BOOLEAN
   sorts BIT
   opns 0 (∗! constructor ∗) : → BIT
         1 : → BIT
   eqns
      ofsort BIT
      1 = 0;
endtype
\end{verbatim}

**B.3 Library TYPES**

This library defines several data types modeling different sizes of bit vectors and types required for the control of the DES.

\begin{verbatim}
(* ──────────────────────────────────────────────────────────────────────────── *)
(* exclusive−or operation on (abstract or concrete) bits *)

(type EXTENDED\_BIT is BIT
   opns _xor_ : BIT, BIT → BIT
eqns
      forall B1, B2 : BIT
      ofsort BIT
      B1 xor B1 = 0;
      B1 xor B2 = 1; (∗ assuming decreasing priority among equations ∗)
endtype

(* ──────────────────────────────────────────────────────────────────────────── *)
\end{verbatim}

\textsuperscript{12}\url{http://cadp.inria.fr/tutorial/#lotos}
Asynchronous Data Encryption Standard

(* data types for counting the iterations of the algorithm *)

type ITERATION AS_RENAMED_NAT is NATURAL renamed by sortnames ITERATION for NAT
endtype

type ITERATION is BOOLEAN, ITERATION AS_RENAMED_NAT
  opns 15 : -> ITERATION
  16 : -> ITERATION
  17 : -> ITERATION
  NEXT : ITERATION -> ITERATION
  LAST : ITERATION -> BOOL
  eqns for all IT : ITERATION
    ofsort ITERATION
      15 = Succ (Succ (Succ (Succ (Succ (9))))));
      16 = Succ (15);
      17 = Succ (16);
    ofsort ITERATION
      NEXT (IT) = (IT + 1) mod 17;
    ofsort BOOL
      LAST (IT) = IT == 16;
endtype

(* data type for the control signals used for the multiplexers *)

type PHASE is BOOLEAN, ITERATION
  sorts PHASE
  opns F (*! constructor *) : -> PHASE (* first iteration *)
    N (*! constructor *) : -> PHASE (* intermediate iteration *)
    L (*! constructor *) : -> PHASE (* last iteration *)
  MUX_DATA : ITERATION -> PHASE
  MUX_K : ITERATION -> PHASE
  DMUX_K : ITERATION -> PHASE
  eqns for all IT : ITERATION, P, P1, P2 : PHASE
    ofsort BOOL
      F = F = true;
      N = N = true;
      L = L = true;
      P1 = P2 = false;
    ofsort PHASE
      MUX_DATA (0) = F;
      (IT == 16) => MUX_DATA (IT) = L;
      MUX_DATA (IT) = N;
    ofsort PHASE
      MUX_K (0) = F;
      MUX_K (IT) = N;
    ofsort PHASE
(IT = 15) \Rightarrow \text{DMUX}_K (IT) = L; \\
\text{DMUX}_K (IT) = N;

endtype

(* data type for the controlling the behavior of the shift register *)

type \text{SHIFT} \textbf{is} \text{BOOLEAN, ITERATION} \\
\textbf{sorts} \text{SHIFT} \\
opns \text{NO} (\text{constructor} \textbf{*)} : \rightarrow \text{SHIFT} (* \text{no shift} \textbf{*)} \\
\text{LS1} (\text{constructor} \textbf{*)} : \rightarrow \text{SHIFT} (* 1 \text{ left shift} \textbf{*)} \\
\text{LS2} (\text{constructor} \textbf{*)} : \rightarrow \text{SHIFT} (* 2 \text{ left shifts} \textbf{*)} \\
\text{RS1} (\text{constructor} \textbf{*)} : \rightarrow \text{SHIFT} (* 1 \text{ right shift} \textbf{*)} \\
\text{RS2} (\text{constructor} \textbf{*)} : \rightarrow \text{SHIFT} (* 2 \text{ right shifts} \textbf{*)} \\
eq =, : \text{SHIFT, SHIFT} \rightarrow \text{BOOL} \\
\text{SHIFT\_CODE} : \text{ITERATION, BOOL} \rightarrow \text{SHIFT}

eqns \\
\textbf{forall} IT : \text{ITERATION}, S, S1, S2 : \text{SHIFT} \\
o\textbf{sort} \text{BOOL} \\
\text{NO} = \text{NO} = \text{true}; \\
\text{LS1} = \text{LS1} = \text{true}; \\
\text{LS2} = \text{LS2} = \text{true}; \\
\text{RS1} = \text{RS1} = \text{true}; \\
\text{RS2} = \text{RS2} = \text{true}; \\
\text{S1} = \text{S2} = \text{false}; \\
o\textbf{sort} \text{SHIFT} \\
(\text{IT} = 0) \Rightarrow \text{SHIFT\_CODE} (\text{IT}, \text{false}) = \text{NO}; \\
(\text{IT} = 1) \Rightarrow \text{SHIFT\_CODE} (\text{IT}, \text{false}) = \text{RS1}; \\
(\text{IT} = 8) \Rightarrow \text{SHIFT\_CODE} (\text{IT}, \text{false}) = \text{RS1}; \\
(\text{IT} = 15) \Rightarrow \text{SHIFT\_CODE} (\text{IT}, \text{false}) = \text{RS1}; \\
\text{SHIFT\_CODE} (\text{IT}, \text{false}) = \text{RS2}; \\
(\text{IT} = 0) \Rightarrow \text{SHIFT\_CODE} (\text{IT}, \text{true}) = \text{LS1}; \\
(\text{IT} = 1) \Rightarrow \text{SHIFT\_CODE} (\text{IT}, \text{true}) = \text{LS1}; \\
(\text{IT} = 8) \Rightarrow \text{SHIFT\_CODE} (\text{IT}, \text{true}) = \text{LS1}; \\
(\text{IT} = 15) \Rightarrow \text{SHIFT\_CODE} (\text{IT}, \text{true}) = \text{LS1}; \\
\text{SHIFT\_CODE} (\text{IT}, \text{true}) = \text{LS2};

endtype

(* data type for 4-bit vectors *)

type \text{BIT4} \textbf{is} \text{EXTENDED\_BIT} \\
\textbf{sorts} \text{BIT4} (\text{constructor} \textbf{*)} : \text{implemented by ADT\_BIT4} \\
opns \text{MK\_4} (\text{constructor} \textbf{*)} : \text{implemented by MK\_4} \\
\text{BIT, BIT, BIT, BIT} \rightarrow \text{BIT4}

endtype

(* data type for 6-bit vectors *)
type BIT6 is EXTENDED_BIT
sorts BIT6
opns
MK_6 (¶ constructor ¶) : BIT, BIT, BIT, BIT, BIT, BIT → BIT6
endtype

(* data type for 32-bit vectors *)

type BIT32 is EXTENDED_BIT, BIT4
sorts BIT32
opns
MK_32 (¶ constructor ¶) : BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT
→ BIT32

(* bitwise xor *)
XOR : BIT32, BIT32 → BIT32
(* concatenation of eight 4-bit vectors to form a 32-bit vector *)
MK_32 : BIT4, BIT4, BIT4, BIT4, BIT4, BIT4, BIT4, BIT4 → BIT32

eqns
ofsorit BIT32
ofsorit BIT32
MK_32 (MK_4 (B1, B2, B3, B4), MK_4 (B5, B6, B7, B8), MK_4 (B9, B10, B11, B12), MK_4 (B13, B14, B15, B16), MK_4 (B17, B18, B19, B20), MK_4 (B21, B22, B23, B24), MK_4 (B25, B26, B27, B28), MK_4 (B29, B30, B31, B32)) =
MK_32 (B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12, B13, B14, B15, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B29, B30, B31, B32);

endtype

(* data type for 48-bit vectors *)

type BIT48 is EXTENDED_BIT, BIT6

sorts BIT48

opns MK_48 (*! constructor *) : BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, ..
\( MK_{48} (A_1 \oplus B_1, A_2 \oplus B_2, A_3 \oplus B_3, A_4 \oplus B_4, A_5 \oplus B_5, A_6 \oplus B_6, A_7 \oplus B_7, A_8 \oplus B_8, \\
A_9 \oplus B_9, A_{10} \oplus B_{10}, A_{11} \oplus B_{11}, A_{12} \oplus B_{12}, A_{13} \oplus B_{13}, A_{14} \oplus B_{14}, A_{15} \oplus B_{15}, A_{16} \oplus B_{16}, \\
A_{17} \oplus B_{17}, A_{18} \oplus B_{18}, A_{19} \oplus B_{19}, A_{20} \oplus B_{20}, A_{21} \oplus B_{21}, A_{22} \oplus B_{22}, A_{23} \oplus B_{23}, A_{24} \oplus B_{24}, \\
A_{25} \oplus B_{25}, A_{26} \oplus B_{26}, A_{27} \oplus B_{27}, A_{28} \oplus B_{28}, A_{29} \oplus B_{29}, A_{30} \oplus B_{30}, A_{31} \oplus B_{31}, A_{32} \oplus B_{32}, \\
A_{33} \oplus B_{33}, A_{34} \oplus B_{34}, A_{35} \oplus B_{35}, A_{36} \oplus B_{36}, A_{37} \oplus B_{37}, A_{38} \oplus B_{38}, A_{39} \oplus B_{39}, A_{40} \oplus B_{40}, \\
A_{41} \oplus B_{41}, A_{42} \oplus B_{42}, A_{43} \oplus B_{43}, A_{44} \oplus B_{44}, A_{45} \oplus B_{45}, A_{46} \oplus B_{46}, A_{47} \oplus B_{47}, A_{48} \oplus B_{48}) = \\
ofsort \ BIT6 \\
\quad 1TO6 \ (MK_{48} (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}, B_{25}, B_{26}, B_{27}, B_{28}, B_{29}, B_{30}, B_{31}, B_{32}, B_{33}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{39}, B_{40}, B_{41}, B_{42}, B_{43}, B_{44}, B_{45}, B_{46}, B_{47}, B_{48})) = MK_6 (B_1, B_2, B_3, B_4, B_5, B_6); \\
ofsort \ BIT6 \\
\quad 7TO12 \ (MK_{48} (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}, B_{25}, B_{26}, B_{27}, B_{28}, B_{29}, B_{30}, B_{31}, B_{32}, B_{33}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{39}, B_{40}, B_{41}, B_{42}, B_{43}, B_{44}, B_{45}, B_{46}, B_{47}, B_{48})) = MK_6 (B_7, B_8, B_9, B_{10}, B_{11}, B_{12}); \\
ofsort \ BIT6 \\
\quad 13TO18 \ (MK_{48} (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}, B_{25}, B_{26}, B_{27}, B_{28}, B_{29}, B_{30}, B_{31}, B_{32}, B_{33}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{39}, B_{40}, B_{41}, B_{42}, B_{43}, B_{44}, B_{45}, B_{46}, B_{47}, B_{48})) = MK_6 (B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}); \\
ofsort \ BIT6 \\
\quad 19TO24 \ (MK_{48} (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}, B_{25}, B_{26}, B_{27}, B_{28}, B_{29}, B_{30}, B_{31}, B_{32}, B_{33}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{39}, B_{40}, B_{41}, B_{42}, B_{43}, B_{44}, B_{45}, B_{46}, B_{47}, B_{48})) = MK_6 (B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}); \\
ofsort \ BIT6 \\
\quad 25TO30 \ (MK_{48} (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}, B_{25}, B_{26}, B_{27}, B_{28}, B_{29}, B_{30}, B_{31}, B_{32}, B_{33}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{39}, B_{40}, B_{41}, B_{42}, B_{43}, B_{44}, B_{45}, B_{46}, B_{47}, B_{48})) = MK_6 (B_{25}, B_{26}, B_{27}, B_{28}, B_{29}, B_{30}, B_{31}, B_{32}, B_{33}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{39}, B_{40}, B_{41}, B_{42}, B_{43}, B_{44}, B_{45}, B_{46}, B_{47}, B_{48});
ofsort BIT6

31TO36 (MK_{48} (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8,
B_9, B_10, B_11, B_12, B_13, B_14, B_15, B_16,
B_17, B_18, B_19, B_20, B_21, B_22, B_23, B_24,
B_25, B_26, B_27, B_28, B_29, B_30, B_31, B_32,
B_33, B_34, B_35, B_36, B_37, B_38, B_39, B_40,
B_41, B_42, B_43, B_44, B_45, B_46, B_47, B_48)) =
MK_6 (B_25, B_26, B_27, B_28, B_29, B_30);

ofsort BIT6

37TO42 (MK_{48} (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8,
B_9, B_10, B_11, B_12, B_13, B_14, B_15, B_16,
B_17, B_18, B_19, B_20, B_21, B_22, B_23, B_24,
B_25, B_26, B_27, B_28, B_29, B_30, B_31, B_32,
B_33, B_34, B_35, B_36, B_37, B_38, B_39, B_40,
B_41, B_42, B_43, B_44, B_45, B_46, B_47, B_48)) =
MK_6 (B_31, B_32, B_33, B_34, B_35, B_36);

ofsort BIT6

43TO48 (MK_{48} (B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8,
B_9, B_10, B_11, B_12, B_13, B_14, B_15, B_16,
B_17, B_18, B_19, B_20, B_21, B_22, B_23, B_24,
B_25, B_26, B_27, B_28, B_29, B_30, B_31, B_32,
B_33, B_34, B_35, B_36, B_37, B_38, B_39, B_40,
B_41, B_42, B_43, B_44, B_45, B_46, B_47, B_48)) =
MK_6 (B_37, B_38, B_39, B_40, B_41, B_42);

endtype

(* data type for 56-bit vectors *)

type BIT56 is EXTENDED_BIT
sorts BIT56
opns

MK_56 (*! constructor *) : BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT,
BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT,
BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT,
BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT,
BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT,
BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT

* left and right shift functions *)
LSHIFT : BIT56 -> BIT56
RSHIFT : BIT56 -> BIT56
eqns
forall B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8,
B_9, B_10, B_11, B_12, B_13, B_14, B_15, B_16,
B_17, B_18, B_19, B_20, B_21, B_22, B_23, B_24,
B_25, B_26, B_27, B_28, B_29, B_30, B_31, B_32,
B33, B34, B35, B36, B37, B38, B39, B40, B41, B42, B43, B44, B45, B46, B47, B48, B49, B50, B51, B52, B53, B54, B55, B56, B57, B58, B59, B60, B61, B62, B63, B64: BIT

ofsort BIT56

LSHIFT (MK_56 (B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12, B13, B14, B15, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B29, B30, B31, B32, B33, B34, B35, B36, B37, B38, B39, B40, B41, B42, B43, B44, B45, B46, B47, B48, B49, B50, B51, B52, B53, B54, B55, B56)) = MK_56 (B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12, B13, B14, B15, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B1, B30, B31, B32, B33, B34, B35, B36, B37, B38, B39, B40, B41, B42, B43, B44, B45, B46, B47, B48, B49, B50, B51, B52, B53, B54, B55, B56, B29);

ofsort BIT56

RSHIFT (MK_56 (B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12, B13, B14, B15, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B1, B30, B31, B32, B33, B34, B35, B36, B37, B38, B39, B40, B41, B42, B43, B44, B45, B46, B47, B48, B49, B50, B51, B52, B53, B54, B55, B56)) = MK_56 (B28, B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12, B13, B14, B15, B16, B17, B18, B19, B20, B21, B22, B23, B24, B25, B26, B27, B28, B56, B29, B30, B31, B32, B33, B34, B35, B36, B37, B38, B39, B40, B41, B42, B43, B44, B45, B46, B47, B48, B49, B50, B51, B52, B53, B54, B55, B56);

datatype

(* data type for 64-bit vectors *)

type BIT64 is EXTENDED_BIT, BIT4, BIT32

sorts BIT64 (*! implemented by ADT_BIT64 printed by ADT_PRINT_BIT64 *)

ops

MK_64 (*! implemented by MK_64 constructor *):

BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BIT, BI
(* concatenation of sixteen 4-bit vectors to form a 64-bit vector *)

MK_64 (*! implemented by CONCAT_BIT4 *) :

BIT4, BIT4, BIT4, BIT4, BIT4, BIT4,
BIT4, BIT4, BIT4, BIT4, BIT4, BIT4, BIT4, BIT4, BIT4 -> BIT64

(* concatenation of two 32-bit vectors to a 64-bit vector *)

MK_64 : BIT32, BIT32 -> BIT64

(* two projection functions from 64-bit vectors to 32-bit vectors *)

1TO32 : BIT64 -> BIT32

33TO64 : BIT64 -> BIT32

eqns

forall B1, B2, B3, B4, B5, B6, B7, B8,
B9, B10, B11, B12, B13, B14, B15, B16,
B17, B18, B19, B20, B21, B22, B23, B24,
B25, B26, B27, B28, B29, B30, B31, B32,
B33, B34, B35, B36, B37, B38, B39, B40,
B41, B42, B43, B44, B45, B46, B47, B48,
B49, B50, B51, B52, B53, B54, B55, B56,
B57, B58, B59, B60, B61, B62, B63, B64:

BIT

ofsort BIT64

MK_64 (MK_4 (B1, B2, B3, B4), MK_4 (B5, B6, B7, B8),
MK_4 (B9, B10, B11, B12), MK_4 (B13, B14, B15, B16),
MK_4 (B17, B18, B19, B20), MK_4 (B21, B22, B23, B24),
MK_4 (B25, B26, B27, B28), MK_4 (B29, B30, B31, B32),
MK_4 (B33, B34, B35, B36), MK_4 (B37, B38, B39, B40),
MK_4 (B41, B42, B43, B44), MK_4 (B45, B46, B47, B48),
MK_4 (B49, B50, B51, B52), MK_4 (B53, B54, B55, B56),
MK_4 (B57, B58, B59, B60), MK_4 (B61, B62, B63, B64)) =

MK_64 (B1, B2, B3, B4, B5, B6, B7, B8,
B9, B10, B11, B12, B13, B14, B15, B16,
B17, B18, B19, B20, B21, B22, B23, B24,
B25, B26, B27, B28, B29, B30, B31, B32),

MK_32 (B33, B34, B35, B36, B37, B38, B39, B40),
MK_32 (B41, B42, B43, B44, B45, B46, B47, B48,
B49, B50, B51, B52, B53, B54, B55, B56,
B57, B58, B59, B60, B61, B62, B63, B64)) =

MK_64 (B1, B2, B3, B4, B5, B6, B7, B8,
B9, B10, B11, B12, B13, B14, B15, B16,
B17, B18, B19, B20, B21, B22, B23, B24,
B25, B26, B27, B28, B29, B30, B31, B32,
B33, B34, B35, B36, B37, B38, B39, B40,
B41, B42, B43, B44, B45, B46, B47, B48,
B49, B50, B51, B52, B53, B54, B55, B56,
B57, B58, B59, B60, B61, B62, B63, B64);
Asynchronous Data Encryption Standard

**B.4 Library PERMUTATION_FUNCTIONS**

This library defines the different permutation functions used by the DES.

```plaintext
type PERMUTATION_FUNCTIONS is BIT32, BIT48, BIT56, BIT64

opns E : BIT32 -> BIT48 (* expansion *)
IP : BIT64 -> BIT64 (* initial permutation *)
IIP : BIT64 -> BIT64 (* inverse initial permutation *)
P : BIT32 -> BIT32 (* permutation *)
PC1 : BIT64 -> BIT56 (* permuted choice 1 *)
PC2 : BIT56 -> BIT48 (* permuted choice 2 *)

eqns
forall B1, B2, B3, B4, B5, B6, B7, B8, B9, B10, B11, B12, B13, B14, B15, B16,
  B17, B18, B19, B20, B21, B22, B23, B24,
  B25, B26, B27, B28, B29, B30, B31, B32,
  B33, B34, B35, B36, B37, B38, B39, B40,
  B41, B42, B43, B44, B45, B46, B47, B48,
  B49, B50, B51, B52, B53, B54, B55, B56,
  B57, B58, B59, B60, B61, B62, B63, B64 :
  BIT

ofsort BIT48
E (MK_32 (B1, B2, B3, B4, B5, B6, B7, B8,
  B9, B10, B11, B12, B13, B14, B15, B16,
  B17, B18, B19, B20, B21, B22, B23, B24,
  B25, B26, B27, B28, B29, B30, B31, B32,
  B33, B34, B35, B36, B37, B38, B39, B40,
  B41, B42, B43, B44, B45, B46, B47, B48,
  B49, B50, B51, B52, B53, B54, B55, B56,
  B57, B58, B59, B60, B61, B62, B63, B64)) =
MK_32 (B1, B2, B3, B4, B5, B6, B7, B8,
  B9, B10, B11, B12, B13, B14, B15, B16,
  B17, B18, B19, B20, B21, B22, B23, B24,
  B25, B26, B27, B28, B29, B30, B31, B32);

ofsort BIT32
1TO32 (MK_64 (B1, B2, B3, B4, B5, B6, B7, B8,
  B9, B10, B11, B12, B13, B14, B15, B16,
  B17, B18, B19, B20, B21, B22, B23, B24,
  B25, B26, B27, B28, B29, B30, B31, B32,
  B33, B34, B35, B36, B37, B38, B39, B40,
  B41, B42, B43, B44, B45, B46, B47, B48,
  B49, B50, B51, B52, B53, B54, B55, B56,
  B57, B58, B59, B60, B61, B62, B63, B64)) =
MK_32 (B1, B2, B3, B4, B5, B6, B7, B8,
  B9, B10, B11, B12, B13, B14, B15, B16,
  B17, B18, B19, B20, B21, B22, B23, B24,
  B25, B26, B27, B28, B29, B30, B31, B32);

ofsort BIT32
33TO64 (MK_64 (B1, B2, B3, B4, B5, B6, B7, B8,
  B9, B10, B11, B12, B13, B14, B15, B16,
  B17, B18, B19, B20, B21, B22, B23, B24,
  B25, B26, B27, B28, B29, B30, B31, B32,
  B33, B34, B35, B36, B37, B38, B39, B40,
  B41, B42, B43, B44, B45, B46, B47, B48,
  B49, B50, B51, B52, B53, B54, B55, B56,
  B57, B58, B59, B60, B61, B62, B63, B64)) =
MK_32 (B33, B34, B35, B36, B37, B38, B39, B40,
  B41, B42, B43, B44, B45, B46, B47, B48,
  B49, B50, B51, B52, B53, B54, B55, B56,
  B57, B58, B59, B60, B61, B62, B63, B64);
```
ofsort BIT32
\[ P \left( MK_{32} \right) = \left( B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}, B_{25}, B_{26}, B_{27}, B_{28}, B_{29}, B_{30}, B_{31}, B_{32} \right) \]

ofsort BIT56
\[ PC1 \left( MK_{64} \right) = \left( B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}, B_{25}, B_{26}, B_{27}, B_{28}, B_{29}, B_{30}, B_{31}, B_{32}, B_{33}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{39}, B_{40}, B_{41}, B_{42}, B_{43}, B_{44}, B_{45}, B_{46}, B_{47}, B_{48}, B_{49}, B_{50}, B_{51}, B_{52}, B_{53}, B_{54}, B_{55}, B_{56}, B_{57}, B_{58}, B_{59}, B_{60}, B_{61}, B_{62}, B_{63}, B_{64} \right) \]

ofsort BIT48
\[ PC2 \left( MK_{56} \right) = \left( B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_{10}, B_{11}, B_{12}, B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{19}, B_{20}, B_{21}, B_{22}, B_{23}, B_{24}, B_{25}, B_{26}, B_{27}, B_{28}, B_{29}, B_{30}, B_{31}, B_{32}, B_{33}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{39}, B_{40}, B_{41}, B_{42}, B_{43}, B_{44}, B_{45}, B_{46}, B_{47}, B_{48}, B_{49}, B_{50}, B_{51}, B_{52}, B_{53}, B_{54}, B_{55}, B_{56} \right) \]
Asynchronous Data Encryption Standard

\[
\begin{align*}
\text{ofsort } & \text{BIT64} \\
\text{IP } & (\text{MK}_{64} (B1, B2, B3, B4, B5, B6, B7, B8, \\
& B9, B10, B11, B12, B13, B14, B15, B16, \\
& B17, B18, B19, B20, B21, B22, B23, B24, \\
& B25, B26, B27, B28, B29, B30, B31, B32, \\
& B33, B34, B35, B36, B37, B38, B39, B40, \\
& B41, B42, B43, B44, B45, B46, B47, B48, \\
& B49, B50, B51, B52, B53, B54, B55, B56, \\
& B57, B58, B59, B60, B61, B62, B63, B64)) = \\
\text{MK}_{64} (B58, B50, B42, B34, B26, B18, B10, B2, \\
& B60, B52, B44, B36, B28, B20, B12, B4, \\
& B62, B54, B46, B38, B30, B22, B14, B6, \\
& B64, B56, B48, B40, B32, B24, B16, B8, \\
& B57, B49, B41, B33, B25, B17, B9, B1, \\
& B59, B51, B43, B35, B27, B19, B11, B3, \\
& B61, B53, B45, B37, B29, B21, B13, B5, \\
& B63, B55, B47, B39, B31, B23, B15, B7)); \\
\text{ofsort } & \text{BIT64} \\
\text{IIIP } & (\text{MK}_{64} (B1, B2, B3, B4, B5, B6, B7, B8, \\
& B9, B10, B11, B12, B13, B14, B15, B16, \\
& B17, B18, B19, B20, B21, B22, B23, B24, \\
& B25, B26, B27, B28, B29, B30, B31, B32, \\
& B33, B34, B35, B36, B37, B38, B39, B40, \\
& B41, B42, B43, B44, B45, B46, B47, B48, \\
& B49, B50, B51, B52, B53, B54, B55, B56, \\
& B57, B58, B59, B60, B61, B62, B63, B64)) = \\
\text{MK}_{64} (B40, B8, B48, B16, B56, B24, B64, B32, \\
& B39, B7, B47, B15, B55, B23, B63, B31, \\
& B38, B6, B46, B14, B54, B22, B62, B30, \\
& B37, B5, B45, B13, B53, B21, B61, B29, \\
& B36, B4, B44, B12, B52, B20, B60, B28, \\
& B35, B3, B43, B11, B51, B19, B59, B27, \\
& B34, B2, B42, B10, B50, B18, B58, B26, \\
& B33, B1, B41, B9, B49, B17, B57, B25)); \\
\text{endtype}
\end{align*}
\]

B.5 Library S_BOX_FUNCTIONS

Contrary to the LNT model, the LOTOS model defines each S-box as a functions, which completely expresses the function of the corresponding S-box, rather than the array defining the S-box. This avoids the definition of additional data types for the array, together with the accessor functions (which are generated automatically by the LNT2LOTOS translator). Precisely, each of the following S-box functions associates to any 6-bit vector the 4-bit value encoding of the corresponding entry in the associated S-box table (see Appendix A.5 for an encoding of these tables). The DES standard defines that for a 6-bit vector \(b_1b_2b_3b_4b_5b_6\), corresponds the entry in the \(b_1b_6\)-th row and the \(b_2b_3b_4b_5\)-th column. The defining equations are ordered such that the right hand sides correspond to the usual reading order of the S-box tables of the standard (rows from top to bottom, and each row from left to right).
The definitions of these functions use premisses, because in the case type \texttt{BIT} implements the abstract singleton type, 1 is no constructor and thus not supported on the left hand side of an equation, e.g., the equation

$S1 \ (MK_6 \ (0, 0, 0, 0, 1, 0) = MK_4 \ (0, 1, 0, 0));$

would be rejected by the LOTOS compiler.

\begin{verbatim}
\textbf{type} S\_BOX\_FUNCTIONS \textbf{is} BIT4, BIT6
\textbf{opns} S1 : BIT6 \rightarrow BIT4
S2 : BIT6 \rightarrow BIT4
S3 : BIT6 \rightarrow BIT4
S4 : BIT6 \rightarrow BIT4
S5 : BIT6 \rightarrow BIT4
S6 : BIT6 \rightarrow BIT4
S7 : BIT6 \rightarrow BIT4
S8 : BIT6 \rightarrow BIT4
\textbf{eqns}
}\textit{(*}
\textit{in the equations below, premisses are used rather than pattern matching because '1' is a constructor for type BIT\_CONCRETE but not for type BIT\_ABSTRACT*)}
\textit{*)}
\textbf{forall} BV6 : BIT6
\textbf{ofsort} BIT4
BV6 = MK_6 \ (0, 0, 0, 0, 0, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (1, 1, 1, 0);
BV6 = MK_6 \ (0, 0, 0, 0, 0, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 0, 0);
BV6 = MK_6 \ (0, 0, 0, 0, 1, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (1, 0, 1, 0);
BV6 = MK_6 \ (0, 0, 0, 1, 0, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 1, 0);
BV6 = MK_6 \ (0, 0, 1, 0, 0, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (1, 0, 0, 1);
BV6 = MK_6 \ (0, 0, 1, 0, 0, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 0, 1, 0);
BV6 = MK_6 \ (0, 0, 1, 0, 1, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 0, 1, 1);
BV6 = MK_6 \ (0, 0, 1, 0, 1, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 0, 1);
BV6 = MK_6 \ (0, 0, 1, 1, 0, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 0, 1);
BV6 = MK_6 \ (0, 0, 1, 1, 0, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 1, 0);
BV6 = MK_6 \ (0, 0, 1, 1, 1, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 1, 1);
BV6 = MK_6 \ (0, 0, 1, 1, 1, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 0, 0, 1);
BV6 = MK_6 \ (0, 1, 0, 0, 0, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 0, 0, 1);
BV6 = MK_6 \ (0, 1, 0, 0, 0, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 0, 1, 0);
BV6 = MK_6 \ (0, 1, 0, 0, 1, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 0, 1, 1);
BV6 = MK_6 \ (0, 1, 0, 0, 1, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 0, 1);
BV6 = MK_6 \ (0, 1, 1, 0, 0, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 0, 1);
BV6 = MK_6 \ (0, 1, 1, 0, 0, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 1, 0);
BV6 = MK_6 \ (0, 1, 1, 0, 1, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 0, 0, 1);
BV6 = MK_6 \ (0, 1, 1, 0, 1, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 1, 1);
BV6 = MK_6 \ (0, 1, 1, 1, 0, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 0, 0, 1);
BV6 = MK_6 \ (0, 1, 1, 1, 0, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 1, 1);
BV6 = MK_6 \ (0, 1, 1, 1, 1, 0) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 0, 0, 1);
BV6 = MK_6 \ (0, 1, 1, 1, 1, 1) \Rightarrow S1 \ (BV6) = MK_4 \ (0, 1, 1, 1);
\end{verbatim}
Asynchronous Data Encryption Standard

\[
\begin{align*}
BV6 &= MK_6 (0, 1, 0, 1, 1, 1) \Rightarrow S1 (BV6) = MK_4 (1, 0, 1, 1); \\
BV6 &= MK_6 (0, 1, 1, 0, 0, 1) \Rightarrow S1 (BV6) = MK_4 (1, 0, 0, 1); \\
BV6 &= MK_6 (0, 1, 1, 0, 1, 1) \Rightarrow S1 (BV6) = MK_4 (0, 1, 0, 1); \\
BV6 &= MK_6 (0, 1, 1, 0, 1, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 1, 1); \\
BV6 &= MK_6 (0, 1, 1, 1, 0, 1) \Rightarrow S1 (BV6) = MK_4 (0, 0, 0, 1); \\
BV6 &= MK_6 (0, 1, 1, 0, 0, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 0, 1, 0, 0) \Rightarrow S1 (BV6) = MK_4 (1, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 0, 0, 1, 0) \Rightarrow S1 (BV6) = MK_4 (1, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 0, 0, 0, 0) \Rightarrow S1 (BV6) = MK_4 (0, 1, 0, 0); \\
BV6 &= MK_6 (0, 1, 0, 1, 0, 1) \Rightarrow S1 (BV6) = MK_4 (1, 0, 1, 0); \\
BV6 &= MK_6 (0, 1, 0, 1, 0, 1) \Rightarrow S1 (BV6) = MK_4 (0, 1, 0, 0); \\
BV6 &= MK_6 (0, 1, 0, 0, 1, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 1, 0); \\
BV6 &= MK_6 (1, 1, 0, 1, 1, 1) \Rightarrow S1 (BV6) = MK_4 (0, 1, 1, 1); \\
BV6 &= MK_6 (1, 1, 1, 0, 1, 1) \Rightarrow S1 (BV6) = MK_4 (1, 0, 1, 0); \\
BV6 &= MK_6 (1, 1, 1, 1, 0, 1) \Rightarrow S1 (BV6) = MK_4 (1, 0, 0, 1); \\
BV6 &= MK_6 (1, 1, 1, 1, 1, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 0, 1); \\
BV6 &= MK_6 (1, 1, 1, 0, 1, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (1, 1, 0, 1, 1, 1) \Rightarrow S1 (BV6) = MK_4 (0, 1, 1, 1); \\
BV6 &= MK_6 (1, 1, 0, 1, 0, 1) \Rightarrow S1 (BV6) = MK_4 (0, 1, 0, 1); \\
BV6 &= MK_6 (1, 1, 0, 0, 1, 1) \Rightarrow S1 (BV6) = MK_4 (0, 0, 1, 1); \\
BV6 &= MK_6 (1, 1, 0, 0, 0, 1) \Rightarrow S1 (BV6) = MK_4 (0, 0, 0, 1); \\
BV6 &= MK_6 (1, 0, 1, 1, 1, 0) \Rightarrow S1 (BV6) = MK_4 (0, 1, 1, 0); \\
BV6 &= MK_6 (1, 0, 1, 1, 0, 0) \Rightarrow S1 (BV6) = MK_4 (0, 1, 0, 0); \\
BV6 &= MK_6 (1, 0, 1, 0, 1, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 1, 0); \\
BV6 &= MK_6 (1, 0, 1, 0, 0, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (1, 0, 0, 1, 1, 1) \Rightarrow S1 (BV6) = MK_4 (0, 0, 1, 1); \\
BV6 &= MK_6 (1, 0, 0, 1, 0, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (1, 0, 0, 0, 1, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (1, 0, 0, 0, 0, 0) \Rightarrow S1 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 1, 1, 1, 1) \Rightarrow S2 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 1, 1, 1, 0) \Rightarrow S2 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 1, 1, 0, 0) \Rightarrow S2 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 1, 0, 1, 0) \Rightarrow S2 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 1, 0, 0, 0) \Rightarrow S2 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 0, 1, 1, 0) \Rightarrow S2 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 0, 1, 0, 0) \Rightarrow S2 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 0, 0, 1, 0) \Rightarrow S2 (BV6) = MK_4 (0, 0, 0, 0); \\
BV6 &= MK_6 (0, 1, 0, 0, 0, 0) \Rightarrow S2 (BV6) = MK_4 (0, 0, 0, 0); \end{align*}
\]
\[ BV6 = MK_6 \{0, 1, 1, 0, 0\} \Rightarrow S2 (BV6) = MK_4 \{0, 1, 0, 1\}; \]
\[ BV6 = MK_6 \{0, 1, 1, 1, 0\} \Rightarrow S2 (BV6) = MK_4 \{1, 0, 1, 0\}; \]
\[ BV6 = MK_6 \{0, 0, 0, 0, 1\} \Rightarrow S2 (BV6) = MK_4 \{0, 0, 1, 0\}; \]
\[ BV6 = MK_6 \{0, 0, 1, 1, 1\} \Rightarrow S2 (BV6) = MK_4 \{1, 1, 0, 1\}; \]
\[ BV6 = MK_6 \{0, 0, 0, 1, 1\} \Rightarrow S2 (BV6) = MK_4 \{0, 1, 0, 1\}; \]
\[ BV6 = MK_6 \{0, 0, 1, 0, 1\} \Rightarrow S2 (BV6) = MK_4 \{1, 0, 0, 0\}; \]
\[ BV6 = MK_6 \{0, 1, 1, 1, 1\} \Rightarrow S2 (BV6) = MK_4 \{1, 1, 1, 0\}; \]

**ofsort BIT4**

\[ BV6 = MK_6 \{0, 0, 0, 0, 0\} \Rightarrow S3 (BV6) = MK_4 \{1, 0, 1, 0\}; \]
Asynchronous Data Encryption Standard

$BV_6 = MK_6 \ (0, 0, 0, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 0, 0, 0)$;

$BV_6 = MK_6 \ (0, 0, 0, 1, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (1, 0, 0, 1)$;

$BV_6 = MK_6 \ (0, 0, 0, 1, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (1, 1, 1, 0)$;

$BV_6 = MK_6 \ (0, 0, 1, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 1, 0, 0)$;

$BV_6 = MK_6 \ (0, 0, 1, 0, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (1, 0, 1, 1)$;

$BV_6 = MK_6 \ (0, 0, 1, 1, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (1, 0, 1, 1)$;

$BV_6 = MK_6 \ (0, 0, 1, 1, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (1, 0, 1, 1)$;

$BV_6 = MK_6 \ (0, 1, 0, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 0, 0, 0)$;

$BV_6 = MK_6 \ (0, 1, 0, 0, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 1, 0, 0)$;

$BV_6 = MK_6 \ (0, 1, 0, 1, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 1, 0, 0)$;

$BV_6 = MK_6 \ (0, 1, 0, 1, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 1, 0, 0)$;

$BV_6 = MK_6 \ (0, 1, 1, 0, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 1, 0, 0)$;

$BV_6 = MK_6 \ (0, 1, 1, 0, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 1, 0, 0)$;

$BV_6 = MK_6 \ (0, 1, 1, 1, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 1, 0, 0)$;

$BV_6 = MK_6 \ (0, 1, 1, 1, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 1, 0, 0)$;

$BV_6 = MK_6 \ (1, 0, 0, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 0, 1, 0)$;

$BV_6 = MK_6 \ (1, 0, 0, 0, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 0, 1, 0)$;

$BV_6 = MK_6 \ (1, 0, 0, 1, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 0, 1, 0)$;

$BV_6 = MK_6 \ (1, 0, 0, 1, 1, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 0, 1, 0)$;

$BV_6 = MK_6 \ (1, 0, 1, 0, 0, 0) \Rightarrow S_3 \ (BV_6) = MK_4 \ (0, 0, 1, 0)$;
BV6 = MK_6 \( (1, 0, 1, 0, 1, 0) \) \( \Rightarrow \) S3 \( (BV6) = MK_4 \( (1, 0, 0, 1) \);
BV6 = MK_6 \( (1, 0, 1, 0, 1, 1) \) \( \Rightarrow \) S3 \( (BV6) = MK_4 \( (1, 0, 0, 0) \);
BV6 = MK_6 \( (1, 0, 1, 1, 0, 0) \) \( \Rightarrow \) S3 \( (BV6) = MK_4 \( (0, 1, 1, 1) \);
BV6 = MK_6 \( (1, 1, 0, 0, 1, 0) \) \( \Rightarrow \) S3 \( (BV6) = MK_4 \( (0, 1, 0, 0) \);
BV6 = MK_6 \( (1, 1, 1, 0, 1, 1) \) \( \Rightarrow \) S3 \( (BV6) = MK_4 \( (0, 1, 1, 0) \);
BV6 = MK_6 \( (1, 1, 1, 1, 0, 0) \) \( \Rightarrow \) S3 \( (BV6) = MK_4 \( (0, 0, 1, 0) \);
BV6 = MK_6 \( (1, 1, 1, 1, 0, 1) \) \( \Rightarrow \) S3 \( (BV6) = MK_4 \( (0, 0, 0, 0) \);

\textbf{ofsort BIT4}

BV6 = MK_6 \( (0, 0, 0, 0, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 1, 0, 0) \);
BV6 = MK_6 \( (0, 0, 0, 1, 0, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 1, 0, 1) \);
BV6 = MK_6 \( (0, 0, 0, 1, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 0, 0) \);
BV6 = MK_6 \( (0, 0, 0, 1, 1, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 0, 1) \);
BV6 = MK_6 \( (0, 0, 1, 0, 0, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 1, 0, 1) \);
BV6 = MK_6 \( (0, 0, 1, 1, 0, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 1, 0, 0) \);
BV6 = MK_6 \( (0, 0, 1, 1, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 1, 0, 1) \);
BV6 = MK_6 \( (0, 0, 1, 1, 1, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 0, 1, 1) \);
BV6 = MK_6 \( (0, 1, 0, 0, 0, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 0, 1) \);
BV6 = MK_6 \( (0, 1, 0, 0, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 1, 0) \);
BV6 = MK_6 \( (0, 1, 0, 1, 0, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 1, 1, 0) \);
BV6 = MK_6 \( (0, 1, 1, 0, 0, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 1, 1, 1) \);
BV6 = MK_6 \( (0, 1, 1, 0, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 0, 1, 1) \);
BV6 = MK_6 \( (0, 1, 1, 1, 0, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 1, 1) \);
BV6 = MK_6 \( (0, 1, 1, 1, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 1, 0, 1) \);
BV6 = MK_6 \( (0, 1, 1, 1, 1, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 1, 1, 1) \);
BV6 = MK_6 \( (0, 1, 1, 0, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 0, 1, 0) \);
BV6 = MK_6 \( (0, 1, 1, 0, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 0, 0, 1) \);
BV6 = MK_6 \( (0, 1, 1, 1, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 0, 0) \);
BV6 = MK_6 \( (0, 1, 1, 1, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 0, 1) \);
BV6 = MK_6 \( (0, 1, 1, 1, 1, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 1, 0) \);
BV6 = MK_6 \( (1, 0, 0, 0, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 0, 0, 0) \);
BV6 = MK_6 \( (1, 0, 0, 0, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 0, 1, 0) \);
BV6 = MK_6 \( (1, 0, 0, 1, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 0, 0, 1) \);
BV6 = MK_6 \( (1, 0, 0, 1, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 1, 0, 0) \);
BV6 = MK_6 \( (1, 0, 0, 1, 1, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 1, 0, 1) \);
BV6 = MK_6 \( (1, 0, 1, 0, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 0, 0) \);
BV6 = MK_6 \( (1, 0, 1, 0, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 1, 0) \);
BV6 = MK_6 \( (1, 0, 1, 1, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 1, 0) \);
BV6 = MK_6 \( (1, 1, 0, 0, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 0, 0, 1) \);
BV6 = MK_6 \( (1, 1, 0, 1, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 0, 1, 0) \);
BV6 = MK_6 \( (1, 1, 0, 1, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 1, 0, 0) \);
BV6 = MK_6 \( (1, 1, 1, 0, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (0, 1, 0, 1) \);
BV6 = MK_6 \( (1, 1, 1, 0, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 1, 0) \);
BV6 = MK_6 \( (1, 1, 1, 1, 0, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 1, 0) \);
BV6 = MK_6 \( (1, 1, 1, 1, 1, 0) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 1, 0) \);
BV6 = MK_6 \( (1, 1, 1, 1, 1, 1) \) \( \Rightarrow \) S4 \( (BV6) = MK_4 \( (1, 0, 1, 0) \);
BV6 = MK₆ (1, 1, 0, 0, 0, 0) \Rightarrow S₄ (BV6) = MK₄ (1, 1, 1, 1);
BV6 = MK₆ (1, 1, 0, 0, 1, 0) \Rightarrow S₄ (BV6) = MK₄ (0, 0, 1, 1);
BV6 = MK₆ (1, 1, 0, 1, 0, 0) \Rightarrow S₄ (BV6) = MK₄ (0, 0, 1, 1);
BV6 = MK₆ (1, 1, 0, 1, 1, 0) \Rightarrow S₄ (BV6) = MK₄ (1, 1, 1, 0);
BV6 = MK₆ (1, 1, 1, 0, 0, 0) \Rightarrow S₄ (BV6) = MK₄ (0, 0, 0, 0);
BV6 = MK₆ (1, 1, 1, 0, 1, 0) \Rightarrow S₄ (BV6) = MK₄ (1, 1, 0, 0);
BV6 = MK₆ (1, 1, 1, 1, 0, 0) \Rightarrow S₄ (BV6) = MK₄ (0, 0, 1, 1);
BV6 = MK₆ (1, 1, 1, 1, 1, 0) \Rightarrow S₄ (BV6) = MK₄ (1, 1, 0, 1);
BV6 = MK₆ (1, 1, 1, 1, 1, 1) \Rightarrow S₄ (BV6) = MK₄ (1, 1, 1, 1);

**ofsort BIT₄**

BV6 = MK₆ (0, 0, 0, 0, 0, 0) \Rightarrow S₅ (BV6) = MK₅ (0, 0, 1, 0);
BV6 = MK₆ (0, 0, 0, 0, 1, 0) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 0, 1, 0, 0, 0) \Rightarrow S₅ (BV6) = MK₅ (0, 0, 1, 0);
BV6 = MK₆ (0, 0, 1, 0, 1, 0) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 0, 1, 1, 0, 0) \Rightarrow S₅ (BV6) = MK₅ (0, 0, 1, 0);
BV6 = MK₆ (0, 1, 0, 0, 0, 0) \Rightarrow S₅ (BV6) = MK₅ (0, 0, 0, 0);
BV6 = MK₆ (0, 1, 0, 0, 1, 0) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 1);
BV6 = MK₆ (0, 1, 0, 1, 0, 0) \Rightarrow S₅ (BV6) = MK₅ (0, 0, 1, 1);
BV6 = MK₆ (0, 1, 0, 1, 1, 0) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 1);
BV6 = MK₆ (0, 1, 1, 0, 0, 0) \Rightarrow S₅ (BV6) = MK₅ (0, 0, 0, 0);
BV6 = MK₆ (0, 1, 1, 0, 1, 0) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 1, 0);
BV6 = MK₆ (0, 1, 1, 1, 0, 0) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 1, 0);
BV6 = MK₆ (0, 1, 1, 1, 1, 0) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 1, 1);
BV6 = MK₆ (0, 0, 0, 0, 0, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 0, 0, 0, 1, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 0, 0, 1, 0, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 0, 1, 0, 0, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 0, 1, 0, 1, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 0, 1, 1, 0, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 1, 0, 0, 0, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 1, 0, 0, 1, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 1, 0, 1, 0, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 1, 0, 1, 1, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 1, 1, 0, 0, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 1, 1, 0, 1, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 1, 1, 1, 0, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
BV6 = MK₆ (0, 1, 1, 1, 1, 1) \Rightarrow S₅ (BV6) = MK₅ (1, 1, 0, 0);
\[ BV6 = MK_6 (0, 1, 0, 1, 1, 1) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 1, 0);
\[ BV6 = MK_6 (0, 1, 1, 0, 1, 1) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 1, 1);
\[ BV6 = MK_6 (0, 1, 1, 1, 0, 1) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 1, 0);
\[ BV6 = MK_6 (0, 1, 1, 1, 1, 0) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 1, 1);
\[ BV6 = MK_6 (1, 0, 0, 1, 0, 1) \Rightarrow S5 \] (BV6) = MK_4 (0, 0, 1, 0);
\[ BV6 = MK_6 (1, 0, 1, 0, 1, 0) \Rightarrow S5 \] (BV6) = MK_4 (0, 0, 1, 1);
\[ BV6 = MK_6 (1, 0, 1, 1, 0, 0) \Rightarrow S5 \] (BV6) = MK_4 (0, 0, 1, 0);
\[ BV6 = MK_6 (1, 0, 1, 1, 1, 0) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 0, 0);
\[ BV6 = MK_6 (1, 1, 0, 0, 1, 0) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 0, 0);
\[ BV6 = MK_6 (1, 1, 0, 1, 0, 0) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 0, 1);
\[ BV6 = MK_6 (1, 1, 1, 0, 0, 0) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 0, 0);
\[ BV6 = MK_6 (1, 1, 1, 0, 1, 0) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 0, 1);
\[ BV6 = MK_6 (1, 1, 1, 1, 0, 0) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 0, 0);
\[ BV6 = MK_6 (1, 1, 1, 1, 1, 0) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 0, 0);
\[ BV6 = MK_6 (1, 1, 1, 1, 1, 1) \Rightarrow S5 \] (BV6) = MK_4 (1, 0, 0, 0);

\[ ofsort ~ BIT4 \]
\[ BV6 = MK_6 (0, 0, 0, 0, 0, 0) \Rightarrow S6 \] (BV6) = MK_4 (1, 1, 0, 0);
\[ BV6 = MK_6 (0, 0, 0, 0, 1, 0) \Rightarrow S6 \] (BV6) = MK_4 (0, 0, 0, 1);
\[ BV6 = MK_6 (0, 0, 0, 1, 0, 0) \Rightarrow S6 \] (BV6) = MK_4 (1, 0, 1, 0);
\[ BV6 = MK_6 (0, 0, 1, 0, 0, 0) \Rightarrow S6 \] (BV6) = MK_4 (1, 1, 1, 0);
\[ BV6 = MK_6 (0, 0, 1, 0, 1, 0) \Rightarrow S6 \] (BV6) = MK_4 (1, 1, 0, 0);
\[ BV6 = MK_6 (0, 1, 0, 0, 0, 0) \Rightarrow S6 \] (BV6) = MK_4 (0, 0, 0, 1);
\[ BV6 = MK_6 (0, 1, 0, 1, 0, 0) \Rightarrow S6 \] (BV6) = MK_4 (0, 0, 1, 0);
\[ BV6 = MK_6 (0, 1, 1, 0, 0, 0) \Rightarrow S6 \] (BV6) = MK_4 (0, 1, 0, 0);
\[ BV6 = MK_6 (0, 1, 1, 0, 1, 0) \Rightarrow S6 \] (BV6) = MK_4 (0, 1, 0, 1);
\[ BV6 = MK_6 (0, 1, 1, 1, 0, 0) \Rightarrow S6 \] (BV6) = MK_4 (1, 1, 0, 0);
\[ BV6 = MK_6 (0, 1, 1, 1, 1, 0) \Rightarrow S6 \] (BV6) = MK_4 (1, 1, 0, 1);
\[ BV6 = MK_6 (0, 1, 1, 1, 1, 1) \Rightarrow S6 \] (BV6) = MK_4 (1, 1, 1, 1);
Asynchronous Data Encryption Standard

\[ \text{ofsort BIT4} \]

\[ BV6 = \text{MK}_6 (0, 0, 0, 0, 0, 0) \implies S7 (BV6) = \text{MK}_4 (0, 1, 0, 0); \]

\[ BV6 = \text{MK}_6 (0, 0, 0, 0, 0, 1) \implies S6 (BV6) = \text{MK}_4 (0, 1, 0, 0); \]

\[ BV6 = \text{MK}_6 (0, 0, 0, 0, 1, 0) \implies S6 (BV6) = \text{MK}_4 (0, 1, 0, 1); \]

\[ BV6 = \text{MK}_6 (0, 0, 0, 1, 0, 0) \implies S6 (BV6) = \text{MK}_4 (0, 1, 1, 0); \]

\[ BV6 = \text{MK}_6 (0, 0, 1, 0, 0, 0) \implies S6 (BV6) = \text{MK}_4 (0, 1, 1, 1); \]

\[ BV6 = \text{MK}_6 (0, 1, 0, 0, 0, 0) \implies S6 (BV6) = \text{MK}_4 (1, 0, 0, 0); \]

\[ BV6 = \text{MK}_6 (0, 1, 0, 0, 0, 1) \implies S6 (BV6) = \text{MK}_4 (1, 0, 0, 1); \]

\[ BV6 = \text{MK}_6 (0, 1, 0, 1, 0, 0) \implies S6 (BV6) = \text{MK}_4 (1, 0, 1, 0); \]

\[ BV6 = \text{MK}_6 (0, 1, 0, 1, 0, 1) \implies S6 (BV6) = \text{MK}_4 (1, 0, 1, 1); \]

\[ BV6 = \text{MK}_6 (0, 1, 1, 0, 0, 0) \implies S6 (BV6) = \text{MK}_4 (1, 1, 0, 0); \]

\[ BV6 = \text{MK}_6 (0, 1, 1, 0, 0, 1) \implies S6 (BV6) = \text{MK}_4 (1, 1, 0, 1); \]

\[ BV6 = \text{MK}_6 (0, 1, 1, 1, 0, 0) \implies S6 (BV6) = \text{MK}_4 (1, 1, 1, 0); \]

\[ BV6 = \text{MK}_6 (0, 1, 1, 1, 0, 1) \implies S6 (BV6) = \text{MK}_4 (1, 1, 1, 1); \]

\[ BV6 = \text{MK}_6 (0, 1, 1, 1, 1) \implies S6 (BV6) = \text{MK}_4 (0, 0, 0, 0); \]

\[ BV6 = \text{MK}_6 (0, 1, 1, 1, 1, 0) \implies S6 (BV6) = \text{MK}_4 (0, 0, 1, 0); \]

\[ BV6 = \text{MK}_6 (0, 1, 1, 1, 1, 1) \implies S6 (BV6) = \text{MK}_4 (0, 1, 0, 0); \]

\[ BV6 = \text{MK}_6 (1, 0, 0, 1, 1) \implies S6 (BV6) = \text{MK}_4 (0, 1, 1, 0); \]

\[ BV6 = \text{MK}_6 (1, 0, 1, 0, 1) \implies S6 (BV6) = \text{MK}_4 (1, 0, 0, 0); \]

\[ BV6 = \text{MK}_6 (1, 0, 1, 1, 0) \implies S6 (BV6) = \text{MK}_4 (1, 0, 1, 0); \]

\[ BV6 = \text{MK}_6 (1, 1, 0, 0, 1) \implies S6 (BV6) = \text{MK}_4 (1, 1, 0, 0); \]

\[ BV6 = \text{MK}_6 (1, 1, 0, 1, 0) \implies S6 (BV6) = \text{MK}_4 (1, 1, 0, 1); \]

\[ BV6 = \text{MK}_6 (1, 1, 1, 0, 0) \implies S6 (BV6) = \text{MK}_4 (1, 1, 1, 0); \]

\[ BV6 = \text{MK}_6 (1, 1, 1, 0, 1) \implies S6 (BV6) = \text{MK}_4 (1, 1, 1, 1); \]

\[ BV6 = \text{MK}_6 (1, 1, 1, 1) \implies S6 (BV6) = \text{MK}_4 (0, 0, 0, 0); \]

\[ BV6 = \text{MK}_6 (1, 1, 1, 1, 0) \implies S6 (BV6) = \text{MK}_4 (0, 0, 1, 0); \]

\[ BV6 = \text{MK}_6 (1, 1, 1, 1, 1) \implies S6 (BV6) = \text{MK}_4 (0, 1, 0, 0); \]

\[ BV6 = \text{MK}_6 (1, 1, 1, 1, 1, 0) \implies S6 (BV6) = \text{MK}_4 (1, 0, 0, 0); \]

\[ BV6 = \text{MK}_6 (1, 1, 1, 1, 1, 1) \implies S6 (BV6) = \text{MK}_4 (1, 0, 1, 0); \]

\[ BV6 = \text{MK}_6 (1, 1, 1, 1, 1, 1, 0) \implies S6 (BV6) = \text{MK}_4 (1, 1, 0, 0); \]

\[ BV6 = \text{MK}_6 (1, 1, 1, 1, 1, 1, 1) \implies S6 (BV6) = \text{MK}_4 (1, 1, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 0, 0, 1, 0) \implies S7 (BV6) = MK_4 (1, 0, 1, 1); \]
\[ BV6 = MK_6 (0, 0, 0, 1, 0, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 0, 0, 1, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 1, 1, 0, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 0, 0, 1, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 1, 0, 0, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 0, 0, 0, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 1, 0, 1, 1, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 1, 0, 1, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 1, 0, 1, 0, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 1, 1, 0, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 0, 1, 0, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
\[ BV6 = MK_6 (0, 0, 0, 0, 0, 0) \implies S7 (BV6) = MK_4 (0, 0, 1, 0); \]
Asynchronous Data Encryption Standard

\[ BV6 = MK_6 \ (1, \ 0, \ 1, \ 0, \ 1, \ 1) \Rightarrow S7 \ (BV6) = MK_4 \ (0, \ 0, \ 0, \ 0); \]
\[ BV6 = MK_6 \ (1, \ 0, \ 1, \ 1, \ 0, \ 1) \Rightarrow S7 \ (BV6) = MK_4 \ (1, \ 0, \ 1, \ 0); \]
\[ BV6 = MK_6 \ (1, \ 0, \ 1, \ 1, \ 1, \ 1) \Rightarrow S7 \ (BV6) = MK_4 \ (0, \ 1, \ 1, \ 1); \]
\[ BV6 = MK_6 \ (1, \ 0, \ 0, \ 0, \ 1, \ 0) \Rightarrow S7 \ (BV6) = MK_4 \ (1, \ 0, \ 0, \ 1); \]
\[ BV6 = MK_6 \ (1, \ 1, \ 0, \ 1, \ 1, \ 1) \Rightarrow S7 \ (BV6) = MK_4 \ (0, \ 1, \ 1, \ 1); \]
\[ BV6 = MK_6 \ (1, \ 1, \ 0, \ 1, \ 0, \ 1) \Rightarrow S7 \ (BV6) = MK_4 \ (0, \ 0, \ 1, \ 0); \]
\[ BV6 = MK_6 \ (1, \ 1, \ 1, \ 0, \ 1, \ 1) \Rightarrow S7 \ (BV6) = MK_4 \ (1, \ 1, \ 1, \ 0); \]
\[ BV6 = MK_6 \ (1, \ 1, \ 1, \ 1, \ 0, \ 1) \Rightarrow S7 \ (BV6) = MK_4 \ (0, \ 0, \ 1, \ 1); \]
\[ BV6 = MK_6 \ (1, \ 1, \ 1, \ 1, \ 1, \ 1) \Rightarrow S7 \ (BV6) = MK_4 \ (1, \ 1, \ 0, \ 0); \]

**ofsort BIT4**

\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 0, \ 0, \ 1) \Rightarrow S8 \ (BV6) = MK_4 \ (1, \ 1, \ 1, \ 0); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 0, \ 1, \ 0) \Rightarrow S8 \ (BV6) = MK_4 \ (0, \ 0, \ 1, \ 1); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 1, \ 0, \ 0) \Rightarrow S8 \ (BV6) = MK_4 \ (1, \ 1, \ 1, \ 0); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 1, \ 0, \ 0, \ 0) \Rightarrow S8 \ (BV6) = MK_4 \ (0, \ 0, \ 1, \ 0); \]
\[ BV6 = MK_6 \ (0, \ 1, \ 0, \ 0, \ 0, \ 0) \Rightarrow S8 \ (BV6) = MK_4 \ (1, \ 1, \ 0, \ 0); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 0, \ 0, \ 1) \Rightarrow S8 \ (BV6) = MK_4 \ (0, \ 0, \ 1, \ 1); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 1, \ 0, \ 1) \Rightarrow S8 \ (BV6) = MK_4 \ (1, \ 1, \ 0, \ 1); \]
\[ BV6 = MK_6 \ (0, \ 1, \ 0, \ 0, \ 0, \ 1) \Rightarrow S8 \ (BV6) = MK_4 \ (0, \ 1, \ 0, \ 1); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 1, \ 1, \ 0) \Rightarrow S8 \ (BV6) = MK_4 \ (1, \ 1, \ 1, \ 0); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 0, \ 1, \ 1) \Rightarrow S8 \ (BV6) = MK_4 \ (0, \ 0, \ 0, \ 1); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 0, \ 0, \ 1) \Rightarrow S8 \ (BV6) = MK_4 \ (1, \ 1, \ 1, \ 1); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 0, \ 1, \ 0) \Rightarrow S8 \ (BV6) = MK_4 \ (0, \ 0, \ 0, \ 0); \]
\[ BV6 = MK_6 \ (0, \ 0, \ 0, \ 0, \ 0, \ 0) \Rightarrow S8 \ (BV6) = MK_4 \ (0, \ 0, \ 0, \ 0); \]
\[\begin{align*}
BV6 &= MK_6(1, 1, 0, 0, 0, 0) \Rightarrow S8(BV6) = MK_4(0, 0, 0, 0);
BV6 &= MK_6(1, 1, 0, 0, 1, 0) \Rightarrow S8(BV6) = MK_4(1, 1, 0, 0); \\
BV6 &= MK_6(1, 1, 0, 1, 0, 0) \Rightarrow S8(BV6) = MK_4(0, 1, 1, 0); \\
BV6 &= MK_6(1, 1, 0, 1, 1, 0) \Rightarrow S8(BV6) = MK_4(1, 0, 1, 0); \\
BV6 &= MK_6(1, 0, 0, 0, 0, 1) \Rightarrow S8(BV6) = MK_4(0, 0, 1, 0); \\
BV6 &= MK_6(1, 0, 0, 0, 1, 1) \Rightarrow S8(BV6) = MK_4(0, 0, 0, 1); \\
BV6 &= MK_6(1, 0, 0, 1, 0, 1) \Rightarrow S8(BV6) = MK_4(1, 1, 0, 0); \\
BV6 &= MK_6(1, 0, 1, 0, 0, 1) \Rightarrow S8(BV6) = MK_4(1, 0, 1, 1); \\
BV6 &= MK_6(1, 0, 1, 0, 1, 1) \Rightarrow S8(BV6) = MK_4(0, 0, 0, 0); \\
BV6 &= MK_6(1, 0, 1, 1, 0, 1) \Rightarrow S8(BV6) = MK_4(0, 1, 0, 1); \\
BV6 &= MK_6(1, 0, 1, 1, 1, 1) \Rightarrow S8(BV6) = MK_4(0, 1, 1, 0); \\
BV6 &= MK_6(1, 1, 0, 0, 0, 1) \Rightarrow S8(BV6) = MK_4(1, 1, 1, 0); \\
BV6 &= MK_6(1, 1, 0, 0, 1, 1) \Rightarrow S8(BV6) = MK_4(1, 1, 0, 1); \\
BV6 &= MK_6(1, 1, 0, 1, 0, 1) \Rightarrow S8(BV6) = MK_4(0, 0, 1, 1); \\
BV6 &= MK_6(1, 1, 0, 1, 1, 1) \Rightarrow S8(BV6) = MK_4(0, 1, 0, 0); \\
BV6 &= MK_6(1, 1, 1, 0, 0, 1) \Rightarrow S8(BV6) = MK_4(0, 1, 1, 0); \\
BV6 &= MK_6(1, 1, 1, 0, 1, 1) \Rightarrow S8(BV6) = MK_4(1, 0, 0, 0); \\
BV6 &= MK_6(1, 1, 1, 1, 0, 1) \Rightarrow S8(BV6) = MK_4(1, 0, 1, 0); \\
BV6 &= MK_6(1, 1, 1, 1, 1, 1) \Rightarrow S8(BV6) = MK_4(1, 1, 0, 0); \\
\end{align*}\]

**endtype**

### B.6 Library CONTROLLER

Process CONTROLLER is responsible for the generation of the appropriate commands to the shift register of the key path as well as the four multiplexers of the data and key paths. It consists of a parallel composition of six processes: one process per controlled block (multiplexer or shift register) plus a process counting the iterations.

**process** CONTROLLER [CRYPT, CTRL_CL, CTRL_CR, CTRL_CK, CTRL_SHIFT, CTRL_DK] : noexit :=

hide CS in

CTRL_MUX_LR [CS, CTRL_CL] [[CS]]
CTRL_MUX_LR [CS, CTRL_CR] [[CS]]
CTRL_MUX_K [CS, CTRL_CK] [[CS]]
CTRL_SHIFT [CRYPT, CS, CTRL_SHIFT] [[CS]]
CTRL_DMUX_K [CS, CTRL_DK] [[CS]]
COUNTER [CS] (0 of ITERATION)

**endproc**
Asynchronous Data Encryption Standard

(*) COUNTER counts the iterations *)

process COUNTER [CS] (IT:ITERATION) : noexit :=
    CS ! IT;
    COUNTER [CS] (NEXT (IT))
endproc

(*) CTRL_MUX_LR generates 1 F, then 14 Ns, and finally 1 L to control one
* of the two multiplexers in the data path *)

process CTRL_MUX_LR [CS, CTRL] : noexit :=
    CS ? IT:ITERATION;
    CTRL ! MUX_DATA (IT);
    CTRL_MUX_LR [CS, CTRL]
endproc

(*) CTRL_MUX_K generates 1 F and 15 N to control the key path multiplexer *)

process CTRL_MUX_K [CS, CTRL] : noexit :=
    CS ? IT:ITERATION;
    ( [not (LAST (IT))] ->
        CTRL ! MUX_K (IT);
        CTRL_MUX_K [CS, CTRL]
    )
endproc

(*) CTRL_SHIFT controls the shift register *)

process CTRL_SHIFT [CRYPT, CS, CTRL] : noexit :=
    CRYPT ? CRYPT:BOOL;
    CTRL_SHIFT_LOOP [CRYPT, CS, CTRL] (CRYPT)
where
    process CTRL_SHIFT_LOOP [CRYPT, CS, CTRL] (CRYPT:BOOL) : noexit :=
        CS ? IT:ITERATION;
        ( [not (LAST (IT))] ->
            CTRL ! SHIFT_CODE (IT, CRYPT);
            CTRL_SHIFT_LOOP [CRYPT, CS, CTRL] (CRYPT)
        )
    endproc
endproc
(* CTRL_DMUX_K generates 15 N and 1 L to control the double multiplexer *)

process CTRL_DMUX_K [CS, CTRL] : noexit :=
    CS ?IT : ITERATION;
    ( [not (LAST (IT))] ->
        CTRL_DMUX_K( IT );
        CTRL_DMUX_K [ CS, CTRL ]
    )
    []
    [ LAST ( IT ) ] ->
        CTRL_DMUX_K [ CS, CTRL ]
) endproc

B.7 Library KEY_PATH

Process KEY_PATH describes the architecture of the key path, generating the sixteen subkeys required for the sixteen iterations.

process KEY_PATH [KEY, SUBKEY, CTRL_CK, CTRL_SHIFT, CTRL_DK] : noexit :=
    hide FIRST_K, INTERMEDIATE_K in
    PC1 [ KEY, FIRST_K ]
    [[FIRST_K]]
    ( [ hide K, KKK, SK in
        ( SHIFT_REGISTER [CTRL_SHIFT, K, SK]
            | [SK]
            | DUPLICATE_K [CTRL_DK, SK, INTERMEDIATE_K, KKK]
        )
        | [K, KKK]
        | CHOOSE_K [CTRL_CK, FIRST_K, KKK, K]
    )
    [[INTERMEDIATE_K]]
    PC2 [INTERMEDIATE_K, SUBKEY]
)

where

(* PCI breaks the initial 64 bit key into a 56-bit vector *)

process PCI [KEY, FIRST_K] : noexit :=
    KEY ?K64:BIT64;
    FIRST_K IPC1( K64 );
    PCI [ KEY, FIRST_K ]
endproc
Asynchronous Data Encryption Standard

(* SHIFT_REGISTER performs, depending on the iteration, one or two shifts to the left or right shift(s) of a 56 bit word *)

process SHIFT_REGISTER [CTRL, INPUT, OUTPUT] : noexit :=
(
  CTRL ?CTRL:SHIFT;
  exit (CTRL, any BIT56)

  ||
  INPUT ?I56:BIT56;
  exit (any SHIFT, I56)
)

>> accept CTRL : SHIFT, I56 : BIT56 in
(
  [CTRL == NO] ->
  OUTPUT !I56;
  SHIFT_REGISTER [CTRL, INPUT, OUTPUT]

  []
  [CTRL == LS1] ->
  OUTPUT !LSHIFT(I56);
  SHIFT_REGISTER [CTRL, INPUT, OUTPUT]

  []
  [CTRL == LS2] ->
  OUTPUT !LSHIFT(LSHIFT(I56));
  SHIFT_REGISTER [CTRL, INPUT, OUTPUT]

  []
  [CTRL == RS1] ->
  OUTPUT !RSHIFT(I56);
  SHIFT_REGISTER [CTRL, INPUT, OUTPUT]

  []
  [CTRL == RS2] ->
  OUTPUT !RSHIFT(RSHIFT(I56));
  SHIFT_REGISTER [CTRL, INPUT, OUTPUT]
)
endproc

(* CHOOSE_K closes the loop in the key path, by redirecting the input on INPUT to OUTPUT, but for the first iteration where the original key is read on FIRST_IN *)

process CHOOSE_K [CTRL, FIRST_IN, INPUT, OUTPUT] : noexit :=
CTRL ?CTRL:PHASE;
(
  [CTRL == F] ->
  FIRST_IN ?I56:BIT56;
  OUTPUT_K [CTRL, FIRST_IN, INPUT, OUTPUT] (I56)

  []
  [CTRL == N] ->
  INPUT ?I56:BIT56;
  OUTPUT_K [CTRL, FIRST_IN, INPUT, OUTPUT] (I56)
where

```plaintext
process OUTPUT_K [CTRL, FIRST_IN, INPUT, OUTPUT]
    (B56 : BIT56) : noexit :=
    OUTPUT ! B56;
    CHOOSE_K [CTRL, FIRST_IN, INPUT, OUTPUT]
endproc
```

```plaintext
process DUPLICATE_K [CTRL, INPUT, OUTPUT1, OUTPUT2] : noexit :=
    (CTRL ? CTRL: PHASE;
     exit (CTRL, any BIT56)
    |||
    INPUT ? I56 : BIT56;
     exit (any PHASE, I56)
    )
>> accept CTRL : PHASE, I56 : BIT56 in
    (CTRL === N) ->
    (OUTPUT1 ! I56;
     exit
    |||
    OUTPUT2 ! I56;
     exit
    )
>>
    DUPLICATE_K [CTRL, INPUT, OUTPUT1, OUTPUT2]

    (CTRL === L) ->
    OUTPUT1 ! I56;
    DUPLICATE_K [CTRL, INPUT, OUTPUT1, OUTPUT2]
)
endproc
```

```plaintext
process PC2 [KK, SUBKEY] : noexit :=
    KK ? I56 : BIT56;
    SUBKEY ! PC2(I56);
    PC2 [KK, SUBKEY]
endproc
```
endproc

B.8 Library CIPHER

The processes of library CIPHER.lib implement Figure 2 of the standard [17], i.e., they compute

\[ P \left( S_i(E(R_i) + K_i) \right) \]

where \( K_i \) is the \( i \)-th subkey and \( R_i \) is the 32-bit vector handled by the \( i \)-th iteration of the DES.

\textbf{process} CIPHER \[ \langle K, R, PX \rangle \] : \textbf{noexit} :=

\begin{verbatim}
hide ER, IS1, IS2, IS3, IS4, IS5, IS6, IS7, IS8,
    SO1, SO2, SO3, SO4, SO5, SO6, SO7, SO8

in

  (E [R, ER]
   |[ER]|
   XOR_48 [ER, K, IS1, IS2, IS3, IS4, IS5, IS6, IS7, IS8]
  ) |

  [IS1, IS2, IS3, IS4, IS5, IS6, IS7, IS8] |

  (S1 [IS1, SO1]
   |||
   S2 [IS2, SO2]
   |||
   S3 [IS3, SO3]
   |||
   S4 [IS4, SO4]
   |||
   S5 [IS5, SO5]
   |||
   S6 [IS6, SO6]
   |||
   S7 [IS7, SO7]
   |||
   S8 [IS8, SO8]
  ) |

  [SO1, SO2, SO3, SO4, SO5, SO6, SO7, SO8] |

P [SO1, SO2, SO3, SO4, SO5, SO6, SO7, SO8, PX]
\end{verbatim}

where

\begin{verbatim}
(* E expands a 32-bit word to 48-bits using function E *)

process E \[ \langle \text{INPUT, OUTPUT} \rangle \] : \textbf{noexit} :=

INPUT ?132:BIT32;
\end{verbatim}
process XOR_48 [A, B, R1, R2, R3, R4, R5, R6, R7, R8] : noexit :=
   (A ? A48 : BIT48;
    exit (A48, any BIT48)
    ||
    B ? B48 : BIT48;
    exit (any BIT48, B48))
   >> accept A48, B48 : BIT48 in
   SPLIT [A, B, R1, R2, R3, R4, R5, R6, R7, R8] (XOR (A48, B48))
where
   (* the auxiliary process factors the computation of XOR (A48, B48) *)
process SPLIT [A, B, R1, R2, R3, R4, R5, R6, R7, R8] (I48 : BIT48) : noexit :=
   (R1 ! (1TO6 (I48)); exit
    ||
    R2 ! (7TO12 (I48)); exit
    ||
    R3 ! (13TO18 (I48)); exit
    ||
    R4 ! (19TO24 (I48)); exit
    ||
    R5 ! (25TO30 (I48)); exit
    ||
    R6 ! (31TO36 (I48)); exit
    ||
    R7 ! (37TO42 (I48)); exit
    ||
    R8 ! (43TO48 (I48)); exit
   )
   >> XOR_48 [A, B, R1, R2, R3, R4, R5, R6, R7, R8]
endproc
endproc

process S1 [INPUT, OUTPUT] : noexit :=
   INPUT ?I6 : BIT6;
   OUTPUT !S1(I6);
   S1 [INPUT, OUTPUT]
endproc
process S2 [INPUT, OUTPUT] : noexit :=
  INPUT ?I6:BIT6;
  OUTPUT !S2(I6);
  S2 [INPUT, OUTPUT]
endproc

process S3 [INPUT, OUTPUT] : noexit :=
  INPUT ?I6:BIT6;
  OUTPUT !S3(I6);
  S3 [INPUT, OUTPUT]
endproc

process S4 [INPUT, OUTPUT] : noexit :=
  INPUT ?I6:BIT6;
  OUTPUT !S4(I6);
  S4 [INPUT, OUTPUT]
endproc

process S5 [INPUT, OUTPUT] : noexit :=
  INPUT ?I6:BIT6;
  OUTPUT !S5(I6);
  S5 [INPUT, OUTPUT]
endproc

process S6 [INPUT, OUTPUT] : noexit :=
  INPUT ?I6:BIT6;
  OUTPUT !S6(I6);
  S6 [INPUT, OUTPUT]
endproc

process S7 [INPUT, OUTPUT] : noexit :=
  INPUT ?I6:BIT6;
  OUTPUT !S7(I6);
  S7 [INPUT, OUTPUT]
endproc

process S8 [INPUT, OUTPUT] : noexit :=
  INPUT ?I6:BIT6;
OUTPUT !S8(I6);
S8 [INPUT, OUTPUT]

endproc

(* P collects the results of the eight processes S.BOX_i (on IN_i) and
* outputs them in a single transition exit; the permutation P is applied
* in a second step when outputting the result on OUTPUT.

* the additional transition with gate exit could be removed by applying
* the permutation P in a subsequent process, namely XOR_32 *)

process P [IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, OUTPUT] : noexit :=
(
    IN1 ?I1:BIT4;
    exit (I1, any BIT4, any BIT4, any BIT4,
         any BIT4, any BIT4, any BIT4)
    ||
    IN2 ?I2:BIT4;
    exit (any BIT4, I2, any BIT4, any BIT4,
         any BIT4, any BIT4, any BIT4)
    ||
    IN3 ?I3:BIT4;
    exit (any BIT4, any BIT4, I3, any BIT4,
         any BIT4, any BIT4, any BIT4)
    ||
    IN4 ?I4:BIT4;
    exit (any BIT4, any BIT4, any BIT4, I4,
         any BIT4, any BIT4, any BIT4)
    ||
    IN5 ?I5:BIT4;
    exit (any BIT4, any BIT4, any BIT4, any BIT4,
         I5, any BIT4, any BIT4)
    ||
    IN6 ?I6:BIT4;
    exit (any BIT4, any BIT4, any BIT4, any BIT4,
         any BIT4, I6, any BIT4)
    ||
    IN7 ?I7:BIT4;
    exit (any BIT4, any BIT4, any BIT4, any BIT4,
         any BIT4, any BIT4, I7)
    ||
    IN8 ?I8:BIT4;
    exit (any BIT4, any BIT4, any BIT4, any BIT4,
         any BIT4, any BIT4, any BIT4)
)
>> accept I1 : BIT4, I2 : BIT4, I3 : BIT4, I4 : BIT4,
    I5 : BIT4, I6 : BIT4, I7 : BIT4, I8 : BIT4 in
OUTPUT !P(MK_32(I1, I2, I3, I4, I5, I6, I7, I8));
P [IN1, IN2, IN3, IN4, IN5, IN6, IN7, IN8, OUTPUT]

endproc
B.9 Library DATA_PATH

This library defines the process DATA_PATH, performing the 16 iterations of the DES, outputing on gate OUTPUT the result of (de)ciphering the data read on gate DATA using the sequence of 16 subkeys received on gate SUBKEY.

\[
\text{process } \text{DATA_PATH} [\text{DATA, OUTPUT, SUBKEY, CTRL_CL, CTRL_CR}] : \text{noexit} := \\
\text{hide } \text{FIRST_L, FIRST_R, OUTPUT_L, OUTPUT_R in} \\
( \\
\text{IP } [\text{DATA, FIRST_L, FIRST_R}] \\
| [\text{FIRST_L, FIRST_R}] \\
| [\text{hide CL,XR, CR,FX, FX,XR, XR,CR, CR_CL in} \\
| \text{CHOOSE_L } [\text{CTRL_CL, FIRST_L, CR_CL, CL,XR, OUTPUT_L}] \\
| [\text{CR_CL}] \\
| \text{CHOOSE_R } [\text{CTRL_CR, FIRST_R, XR,CR, CR_CL, CR,FX, OUTPUT_R}] \\
| [\text{CL,XR, CR,FX, XR_CR}] \\
| \text{CIPHER } [\text{SUBKEY, CR,FX, FX,XR}] \\
| [\text{FX,XR}] \\
| \text{XOR_32 } [\text{CL,XR, FX,XR, XR_CR}] \\
) \\
| [\text{OUTPUT_L, OUTPUT_R}] \\
\text{IIP } [\text{OUTPUT_L, OUTPUT_R, OUTPUT}] \\
\text{where} \\
\text{(* IP applies the initial permutation IP to the initial 64-bit vector} \\
\text{* received on gate DATA and breaks the resulting 64-bit vector into two} \\
\text{* 32-bit vectors L and R *)} \\
\text{process } \text{IP } [\text{DATA, FIRST_L, FIRST_R}] : \text{noexit} := \\
\text{DATA } ?164:BIT64; \\
\text{SPLIT_LR } [\text{DATA, FIRST_L, FIRST_R}] (\text{IP (164)}) \\
\text{where} \\
\text{(* the auxiliary process avoids to compute IP (164) twice *)} \\
\text{process } \text{SPLIT_LR } [\text{DATA, FIRST_L, FIRST_R}] (\text{B64 : BIT64}) : \text{noexit} := \\
( \\
\text{FIRST_L } !1TO32(B64); \\
\text{exit} \\
| \text{FIRST_R } !33TO64(B64); \\
\text{exit} \\
) \\
\text{>> } \text{IP } [\text{DATA, FIRST_L, FIRST_R}]
\]
CHOOSEₗ reads a 32-bit vector from INPUT and outputs to OUTPUT, but for the first iteration, where it reads from FIRST_IN, and the last iteration, where it outputs to LAST_OUT *)

process CHOOSEₗ [CTRL, FIRST_IN, INPUT, OUTPUT, LAST_OUT] : noexit :=
CTRL ?CTRL:PHASE;
  ( [CTRL == F] ->
    FIRST_IN ?L32:BIT32;
    OUTPUT !L32;
    CHOOSEₗ [CTRL, FIRST_IN, INPUT, OUTPUT, LAST_OUT]
  )
  [CTRL == N] ->
    INPUT ?L32:BIT32;
    OUTPUT !L32;
    CHOOSEₗ [CTRL, FIRST_IN, INPUT, OUTPUT, LAST_OUT]
  )
  [CTRL == L] ->
    INPUT ?L32:BIT32;
    LAST_OUT !L32;
    CHOOSEₗ [CTRL, FIRST_IN, INPUT, OUTPUT, LAST_OUT]
)
endproc

CHOOSEₗ reads a 32-bit vector from INPUT and outputs to OUT₁ and OUT₂, but for the first iteration, where it reads from FIRST_IN, and the last one, where it outputs to LAST_OUT *)

process CHOOSEₘ [CTRL, FIRST_IN, INPUT, OUT₁, OUT₂, LAST_OUT]
  : noexit :=
CTRL ?CTRL:PHASE;
  ( [CTRL == F] ->
    FIRST_IN ?R32:BIT32;
    OUTPUT_R [CTRL, FIRST_IN, INPUT, OUT₁, OUT₂, LAST_OUT]
      (R32)
  )
  [CTRL == N] ->
    INPUT ?R32:BIT32;
    OUTPUT_R [CTRL, FIRST_IN, INPUT, OUT₁, OUT₂, LAST_OUT]
      (R32)
  )
  [CTRL == L] ->
    INPUT ?R32:BIT32;
    LAST_OUT !R32;
    CHOOSEₘ [CTRL, FIRST_IN, INPUT, OUT₁, OUT₂, LAST_OUT]
where

process OUTPUT_R [CTRL, FIRST_IN, INPUT, OUT1, OUT2, LAST_OUT] (R32 : BIT32) : noexit :=

OUT1 ! R32;
OUT2 ! R32;

endproc

process XOR_32 [A, B, R] : noexit :=

A ? A32 : BIT32;

exit (A32, any BIT32)

||

B ? B32 : BIT32;

exit (any BIT32, B32)

endproc

process IIP [OUTPUT_L, OUTPUT_R, OUTPUT] : noexit :=

OUTPUT_L ? OL: BIT32;

exit (OL, any BIT32)

||

OUTPUT_R ? OH: BIT32;

exit (any BIT32, OH)

endproc
B.10 Library DES

The library DES.lib factorizes the definition of the architecture of the asynchronous DES, and is shared by all three LOTOS specifications of the DES, namely DES_ABSTRACT.lotos, DES_CONCRETE.lotos, and DES_SAMPLE.lotos.

\[
\text{process} \ \text{DES} \ [\text{CRYPT, KEY, DATA, OUTPUT}] : \text{noexit} := \\
\text{hide} \ \text{SUBKEY, CTRL_CL, CTRL_CR, CTRL_CK, CTRL_SHIFT, CTRL_DK} \ \text{in} \\
( \\
\text{DATA_PATH} \ [\text{DATA, OUTPUT, SUBKEY, CTRL_CL, CTRL_CR}] \\
| \ [\text{SUBKEY}] \\
\text{KEY_PATH} \ [\text{KEY, SUBKEY, CTRL_CK, CTRL_SHIFT, CTRL_DK}] \\
) \\
| \ [\text{CTRL_CL, CTRL_CR, CTRL_CK, CTRL_SHIFT, CTRL_DK}] \\
\text{CONTROLLER} \ [\text{CRYPT, CTRL_CL, CTRL_CR, CTRL_CK, CTRL_SHIFT, CTRL_DK}] \\
\text{endproc}
\]

B.11 Specification with Abstract Bits: DES_ABSTRACT

This LOTOS specification can be used for compositional state space generation and verification. Due to the abstraction, there is no need to close the system with an environment.

\[
\text{specification} \ \text{DES_ABSTRACT} \ [\text{CRYPT, KEY, DATA, OUTPUT}] : \text{noexit} \\
\text{library} \ \text{DES_ABSTRACT, TYPES, PERMUTATION_FUNCTIONS, S_BOX_FUNCTIONS} \\
\text{endlib} \\
\text{behaviour} \\
\text{DES} \ [\text{CRYPT, KEY, DATA, OUTPUT}] \\
\text{where} \\
\text{library} \ \text{DES, CONTROLLER, KEY_PATH, DATA_PATH, CIPHER} \\
\text{endlib} \\
\text{endspec}
\]

B.12 Specification with Concrete Bits: DES_CONCRETE

This LOTOS specification can be used to generate a prototype implementation. It is not suitable for state space generation, because it would have to enumerate over all possible input values, i.e., 64-bit vectors.

\[
\text{specification} \ \text{DES_CONCRETE} \ [\text{CRYPT, KEY, DATA, OUTPUT}] : \text{noexit} \\
\text{library} \ \text{BIT_CONCRETE, TYPES, PERMUTATION_FUNCTIONS, S_BOX_FUNCTIONS}
\]
Asynchronous Data Encryption Standard

endlib

behaviour

DES [CRYPT, KEY, DATA, OUTPUT]

where

library

DES, CONTROLLER, KEY_PATH, DATA_PATH, CIPHER

endlib

endspec

B.13 Specification with Concrete Bits and Environment: DES_SAMPLE

This LOTOS specification can be used to directly (i.e., non compositionally) generate an LTS. After hiding all offers and minimization for branching bisimulation, the generated LTS is the one shown in Figure 2.

specification DES_SAMPLE [CRYPT, KEY, DATA, OUTPUT] : noexit

library

BOOLEAN, NATURAL, BIT_CONCRETE, TYPES,
PERMUTATION_FUNCTIONS, S_BOX_FUNCTIONS
endlib

behaviour

DES [CRYPT, KEY, DATA, OUTPUT]
||[CRYPT, KEY, DATA, OUTPUT]|
ENVIRONMENT [CRYPT, KEY, DATA, OUTPUT]

where

library

DES, CONTROLLER, KEY_PATH, DATA_PATH, CIPHER
endlib

(* ----------------------------------------------------------------------------- *)

type CONSTANTS is BIT64

opns

(* frequently used sample data *)
C_01234567_89ABCDEF : → BIT64
(* frequently used sample key *)
C_13345779_9BBCDFF1 : → BIT64
(* result of ciphering C_01234567_89ABCDEF by C_13345779_9BBCDFF1 *)
C_85E81354_0F0AB405 : → BIT64

eqns

ofsort BIT64
B.14 Specification of Property 4

This LOTOS specification describes an automaton, which is used to verify the correct synchronisation of the data and key paths by equivalence checking.

**specification** PROPERTY_4 [CRYPT, SUBKEY] : noexit

```plaintext
library
  BOOLEAN, NATURAL
endlib

type T is NATURAL

opns
  14 : -> NAT
eqns
```

```plaintext
(* process simulating the environment in order to close the system,
* executing a single encryption of c_01234567_89abcdef with
* c_13345779_9BBCDFF1, and checking the output of the expected result *)
```

```plaintext
process ENVIRONMENT [CRYPT, KEY, DATA, OUTPUT] : noexit :=
  CRYPTO ! true;
  KEY ! C_13345779_9BBCDFF1;
  DATA ! C_01234567_89abcdef;
  (* cipher of C_01234567_89abcdef with C_13345779_9BBCDFF1 *)
  OUTPUT ! C_85E81354_0F0AB405;
  stop
endproc
```

```plaintext
endspec
```
**ofsort** \text{NAT}

\begin{align*}
14 = \text{SUC} & (\text{SUC} (\text{SUC} (\text{SUC} (\text{SUC} (9))))))
\end{align*}

\text{endtype}

\textbf{behaviour}

(* process describing a loop starting with a synchronization on gate \text{CRYPT},
* followed by 16 synchronization on gate \text{SUBKEY}, where the synchronization
* on \text{CRYPT} corresponding to the next iteration can already appear after
* 14 synchronizations on \text{SUBKEY}.
* this process is used to verify the presence of 16 iterations in the DES.
*)

\text{CRYPT};

\text{SIXTEEN\_ITERATIONS} [\text{CRYPT}, \text{SUBKEY}]

\text{where}

\begin{align*}
\text{process} & \text{SIXTEEN\_ITERATIONS} [\text{CRYPT}, \text{SUBKEY}] : \text{noexit} := \\
& \text{FOURTEEN\_ITERATIONS} [\text{SUBKEY}] (0) \\
& \text{SIXTEEN\_ITERATIONS} [\text{CRYPT}, \text{SUBKEY}]
\end{align*}

\text{endproc}

\begin{align*}
\text{process} & \text{FOURTEEN\_ITERATIONS} [\text{SUBKEY}] (N: \text{NAT}) : \text{exit} := \\
& [N = 14] \rightarrow \\
& \text{exit} \\
& [N < 14] \rightarrow \\
& \text{SUBKEY}; \\
& \text{FOURTEEN\_ITERATIONS} [\text{SUBKEY}] (N + 1)
\end{align*}

\text{endproc}

\text{endspec}

\section{C Code for the EXEC/CÆSAR Framework}

To generate a prototype from the LNT or LOTOS model using the EXEC/CÆSAR framework, additional
C code is necessary, namely the main program (for which the example provided with CADP\textsuperscript{13} can

\textsuperscript{13}Precisely, the file \$\text{CADP/src/exec caesar/main.c} included in the CADP toolbox.
be used) and so-called gate functions implementing the interaction with the environment. The DES prototype reads its inputs from the standard input and prints its results to the standard output. Each rendezvous corresponds to one line of input (respectively, output), following the syntax of LOTOS, i.e., “G !O”, where G is a gate name and O is a sequence of characters corresponding to the offer. For convenience, 64-bit vectors (for gates CRYPT, DATA, KEY, and OUTPUT) are represented by a sequence of sixteen hexadecimal digits, and Booleans (for gate CRYPT) by 0 (false) and 1 (true). The prototype can also write its execution trace in the SEQ format to a log file.

C.1 Auxiliary Functions for Reading and Writing

The following auxiliary functions are required to parse and print 64-bit vectors. Each line of input parsed is stored in a variable (local to the module defining the gate functions), because the first gate function called by the prototype might not correspond to the input line, enabling its reuse by another gate function.

```c
#include "caesar_kernel.h"
extern void CAESAR_KERNEL_EXIT();
#define streq(S1,S2) (strcmp ((S1), (S2)) == 0)

static ADT_BIT4 CHAR_TO_BIT4 (C)
char C;
{
    switch (C) {
    case '0':
        return MK_4 (BIT_ZERO (), BIT_ZERO (), BIT_ZERO (), BIT_ZERO ());
    case '1':
        return MK_4 (BIT_ZERO (), BIT_ZERO (), BIT_ZERO (), BIT_ONE ());
    case '2':
        return MK_4 (BIT_ZERO (), BIT_ZERO (), BIT_ONE (), BIT_ZERO ());
    case '3':
        return MK_4 (BIT_ZERO (), BIT_ZERO (), BIT_ONE (), BIT_ONE ());
    case '4':
        return MK_4 (BIT_ZERO (), BIT_ONE (), BIT_ZERO (), BIT_ZERO ());
    case '5':
        return MK_4 (BIT_ZERO (), BIT_ONE (), BIT_ONE (), BIT_ZERO ());
    case '6':
        return MK_4 (BIT_ZERO (), BIT_ONE (), BIT_ONE (), BIT_ONE ());
    case '7':
        return MK_4 (BIT_ZERO (), BIT_ONE (), BIT_ONE (), BIT_ONE ());
    case '8':
        return MK_4 (BIT_ONE (), BIT_ZERO (), BIT_ZERO (), BIT_ZERO ());
    case '9':
        return MK_4 (BIT_ONE (), BIT_ZERO (), BIT_ZERO (), BIT_ZERO ());
    default:
        return MK_4 (BIT_ZERO (), BIT_ZERO (), BIT_ZERO (), BIT_ZERO ());
    }
```

\[\text{http://cadp.inria.fr/man/exhibitor.html#sect3}\]
return MK4 (BIT_ONE (), BIT_ZERO (), BIT_ZERO (), BIT_ONE ());
case 'a': case 'A':
    return MK4 (BIT_ONE (), BIT_ZERO (), BIT_ONE (), BIT_ZERO ());
case 'b': case 'B':
    return MK4 (BIT_ONE (), BIT_ZERO (), BIT_ONE (), BIT_ONE ());
case 'c': case 'C':
    return MK4 (BIT_ONE (), BIT.ZERO (), BIT.ZERO (), BIT.ZERO ());
case 'd': case 'D':
    return MK4 (BIT.ONE (), BIT.ONE (), BIT.ZERO (), BIT.ZERO ());
case 'e': case 'E':
    return MK4 (BIT.ONE (), BIT.ONE (), BIT.ZERO (), BIT.ONE ());
case 'f': case 'F':
    return MK4 (BIT.ONE (), BIT.ONE (), BIT.ONE (), BIT.ONE ());
default:
    CAESAR_KERNEL_EXIT ("cannot convert \%c into a 4-bit vector\n",
                        C);
        /* NOTREACHED */
return MK4 (BIT_ZERO (), BIT.ZERO (), BIT.ZERO (), BIT.ZERO ());
}

/* auxiliary macro to print four bits as a hexadecimal digit */
#define PRINT4_BITS (F, B1, B2, B3, B4) "
{ char N; \n    N = 0; \n    if (((B1) == BIT_ONE ()) N += 8; \n    if (((B2) == BIT_ONE ()) N += 4; \n    if (((B3) == BIT_ONE ()) N += 2; \n    if (((B4) == BIT_ONE ()) N += 1; 
    fprintf ((F), "%x", N); \n}

/* auxiliary macro to access bit number N of the 64-bit vector V */
#define BIT (V, N) (CAESAR_ADTSSTAR_ADTSBIT64 (V).CAESAR_ADTS.# # N # # _MK64)

/* auxiliary function to output a 64-bit vector as 16 hexadecimal digits */
static void PRINT64_BIT_VECTOR (F, V)
FILE *F;
ADT_BIT64 V;
{ PRINT4_BITS (F, BIT (V, 1), BIT (V, 2), BIT (V, 3), BIT (V, 4));
    PRINT4_BITS (F, BIT (V, 5), BIT (V, 6), BIT (V, 7), BIT (V, 8));
PRINT_4_BITS (F, BIT (V, 9), BIT (V, 10), BIT (V, 11), BIT (V, 12));
PRINT_4_BITS (F, BIT (V, 13), BIT (V, 14), BIT (V, 15), BIT (V, 16));
PRINT_4_BITS (F, BIT (V, 17), BIT (V, 18), BIT (V, 19), BIT (V, 20));
PRINT_4_BITS (F, BIT (V, 21), BIT (V, 22), BIT (V, 23), BIT (V, 24));
PRINT_4_BITS (F, BIT (V, 25), BIT (V, 26), BIT (V, 27), BIT (V, 28));
PRINT_4_BITS (F, BIT (V, 29), BIT (V, 30), BIT (V, 31), BIT (V, 32));
PRINT_4_BITS (F, BIT (V, 33), BIT (V, 34), BIT (V, 35), BIT (V, 36));
PRINT_4_BITS (F, BIT (V, 37), BIT (V, 38), BIT (V, 39), BIT (V, 40));
PRINT_4_BITS (F, BIT (V, 41), BIT (V, 42), BIT (V, 43), BIT (V, 44));
PRINT_4_BITS (F, BIT (V, 45), BIT (V, 46), BIT (V, 47), BIT (V, 48));
PRINT_4_BITS (F, BIT (V, 49), BIT (V, 50), BIT (V, 51), BIT (V, 52));
PRINT_4_BITS (F, BIT (V, 53), BIT (V, 54), BIT (V, 55), BIT (V, 56));
PRINT_4_BITS (F, BIT (V, 57), BIT (V, 58), BIT (V, 59), BIT (V, 60));
PRINT_4_BITS (F, BIT (V, 61), BIT (V, 62), BIT (V, 63), BIT (V, 64));

/****************************************************************************
 */
*/ buffer for the gate of the rendezvous in the current line of stdin */
static char CURRENT_GATE[6];

/****************************************************************************
 */
*/ buffer for the offer of the rendezvous in the current line of stdin */
static char CURRENT_OFFER[17];

/****************************************************************************
*/
* if equal to zero, CURRENT_GATE and CURRENT_OFFER contain the gate (resp.
* OFFER) of the rendezvous in the current line of stdin
*/
static char CURRENT_LINE_NOT_READ = 1;

/****************************************************************************
 */
*/ auxiliary function to check whether the current line of the standard
* input corresponds to a rendezvous on gate GATE
*/
int PARSE_INPUT (GATE, OFFER)
char *GATE;
void *OFFER; /* a caster */
{
    int N;

    if (feof (stdin))
        return (0);

    if (CURRENT_LINE_NOT_READ) {
        /* read a line from stdin */
        N = scanf ("%5s!%16[0123456789aAbBcCdDeEfF]\n",
        /*...*/
CURRENT_GATE, CURRENT_OFFER);
    if (N == EOF)
        return (0);
    if (N != 2) {
        CAESAR_KERNEL_EXIT ("incorrect_input_syntax\n");
        CURRENT_LINE_NOT_READ = 0;
    }
    if (!strcmp (CURRENT_GATE, GATE))
        return (0);
    CURRENT_LINE_NOT_READ = 1;

    if (strcmp (GATE, "CRYPT") ) {
        if (strcmp (CURRENT_OFFER, "0") ) {
            *((ADT_BOOL *) OFFER) = ADT_FALSE ( );
        } else if (strcmp (CURRENT_OFFER, "1") ) {
            *((ADT_BOOL *) OFFER) = ADT_TRUE ( );
        } else {
            CAESAR_KERNEL_EXIT ("incorrect_offer\"%s\" for gate\"%s\"\n", CURRENT_OFFER, GATE);
            /* NOTREACHED */
        }
    } else if (strcmp (GATE, "DATA") || strcmp (GATE, "KEY")) {
        /* set remaining digits to '0' */
        for (N = strlen (CURRENT_OFFER) ; N < 16 ; N++)
            CURRENT_OFFER[N] = '0';
        /* concatenation of the 16 characters to form a 64-bit vector */
        *((ADT_BIT64 *) OFFER) =
            CONCAT_BIT4 (CHAR_TO_BIT4 (CURRENT_OFFER[0]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[1]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[2]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[3]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[4]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[5]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[6]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[7]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[8]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[9]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[10]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[11]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[12]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[13]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[14]),
                         CHAR_TO_BIT4 (CURRENT_OFFER[15]));
    } else {
        CAESAR_KERNEL_EXIT ("unknown_gate\"%s\"\n", CURRENT_GATE);
        /* NOTREACHED */
    }

    return (1);
C.2 Gate Functions

Lines typeset in small, light grey font correspond to automatically generated code to check assumptions about the parameters of gate functions and (possibly) log gate function calls to a file. These functions are documented in the header file caesar_kernel.h included in the CADP toolbox.

```c
/* —————————————————————————————————————————*/

int CRYPT (INPUT_MODE, BOOL, OFFER, EOL)
CAESAR_KERNEL_OFFER INPUT_MODE;
char *BOOL;
ADT_BOOL *OFFER;
CAESAR_KERNEL_OFFER EOL;
{
    int CAESAR_STATUS;

    CAESAR_KERNEL_ASSERT_INPUT (INPUT_MODE);
    CAESAR_KERNEL_ASSERT_TYPE (BOOL, "ADT_BOOL");
    CAESAR_KERNEL_ASSERT_EOL (EOL);

    CAESAR_STATUS = PARSE_INPUT ("CRYPT", OFFER);

    CAESAR_KERNEL_LOG_GATE (__func__);
    /* LINTED */
    CAESAR_KERNEL_LOG_OFFER (INPUT_MODE, BOOL, *OFFER, ADT_PRINT_BOOL, CAESAR_STATUS);
    CAESAR_KERNEL_LOG_RESULT (CAESAR_STATUS);

    return (CAESAR_STATUS);
}

/* —————————————————————————————————————————*/

int DATA (INPUT_MODE, BIT64, OFFER, EOL)
CAESAR_KERNEL_OFFER INPUT_MODE;
char *BIT64;
ADT_BIT64 *OFFER;
CAESAR_KERNEL_OFFER EOL;
{
    int CAESAR_STATUS;

    CAESAR_KERNEL_ASSERT_INPUT (INPUT_MODE);
    CAESAR_KERNEL_ASSERT_TYPE (BIT64, "ADT_BIT64");
    CAESAR_KERNEL_ASSERT_EOL (EOL);

    CAESAR_STATUS = PARSE_INPUT ("DATA", OFFER);

    CAESAR_KERNEL_LOG_GATE (__func__);
    /* LINTED */
    CAESAR_KERNEL_LOG_OFFER (INPUT_MODE, BIT64, *OFFER, ADT_PRINT_BIT64, CAESAR_STATUS);
    CAESAR_KERNEL_LOG_RESULT (CAESAR_STATUS);
```
return (CAESAR_STATUS);
}
/* ----------------------------------------------- */

int KEY (INPUT_MODE, BIT64, OFFER, EOL)
CAESAR_KERNEL_OFFER INPUT_MODE;
char *BIT64;
ADT_BIT64 *OFFER;
CAESAR_KERNEL_OFFER EOL;
{
    int CAESAR_STATUS;

    CAESAR_KERNEL_ASSERT_INPUT (INPUT_MODE);
    CAESAR_KERNEL_ASSERT_TYPE (BIT64, "ADT_BIT64");
    CAESAR_KERNEL_ASSERT_EOL (EOL);

    CAESAR_STATUS = PARSE_INPUT ("KEY", OFFER);

    CAESAR_KERNEL_LOG_GATE (__func__);
    /* LINTED */
    CAESAR_KERNEL_LOG_OFFER (INPUT_MODE, BIT64, *OFFER, ADT_PRINT_BIT64, CAESAR_STATUS);
    CAESAR_KERNEL_LOG_RESULT (CAESAR_STATUS);

    return (CAESAR_STATUS);
}
/* ----------------------------------------------- */

int OUTPUT (OUTPUT_MODE, BIT64, OFFER, EOL)
CAESAR_KERNEL_OFFER OUTPUT_MODE;
char *BIT64;
ADT_BIT64 OFFER;
CAESAR_KERNEL_OFFER EOL;
{
    CAESAR_KERNEL_ASSERT_OUTPUT (OUTPUT_MODE);
    CAESAR_KERNEL_ASSERT_TYPE (BIT64, "ADT_BIT64");
    CAESAR_KERNEL_ASSERT_EOL (EOL);

    fprintf (stdout, "OUTPUT!");
    PRINT_64_BIT_VECTOR (stdout, OFFER);
    fprintf (stdout, "\n");

    CAESAR_KERNEL_LOG_GATE (__func__);
    /* LINTED */
    CAESAR_KERNEL_LOG_OFFER (OUTPUT_MODE, BIT64, OFFER, ADT_PRINT_BIT64, 1);
    CAESAR_KERNEL_LOG_RESULT (1);

    return (1);
D  Complete Verification Scenarios for the Asynchronous DES

The complete verification scenario of the asynchronous DES is executed by the following SVL script, using only sequential tools of CADP (i.e., no distributed state space generation is required). The script is also available in the demo example on the CADP website.

The SVL script first compositionally generates the LTS corresponding to the abstract LNT model DES_ABSTRACT.lnt. The script then verifies several properties of the LNT models, using different techniques, such as model checking (for the temporal logic properties of Sections 4 and 5) and equivalence checking, but also the generation of a prototype from the LNT model DES_CONCRETE.lnt and comparing its output for some example data and key with official results. These first steps can be easily adapted to use the LOTOS model instead of the LNT model. Finally, the LNT models are compared (using equivalence checking) with the corresponding LOTOS specifications, which are supposed to be located in a subdirectory called "LOTOS", together with an SVL script called "demo.svl" generating the LTSs for the LOTOS specifications DES_ABSTRACT.lotos and DES_SAMPLE.lotos.

For a description of the syntax of SVL, see the SVL manual page — the most important points being that lines starting with "%%" are Bourne shell commands, and that temporal logic formulas in MCL (Model Checking Language) [14] are directly inlined.

--- Compositional state space generation

--- In order to reduce the time and memory requirements for the generation of the state space and the verification of properties of the model, the domain of 64-bit vectors is reduced to a single value, by simply redefining the data type for bits so as to use a singleton domain:

--- instead of the module "BIT_CONCRETE.lnt", the module "BIT_ABSTRACT.lnt" is used.

--- Each of the three main components of the DES is generated and minimized under the constraints of the already computed parts.

%%% DEFAULT_PROCESS_FILE="DES.lnt"

--- creation of BIT.lnt as a copy of BIT_ABSTRACT.lnt

%%% sed -e ’s/BIT_ABSTRACT/BIT/’ BIT_ABSTRACT.lnt > BIT.lnt

%%% SVL_RECORD_FOR_CLEAN "BIT.lnt"

--- generation and minimization of the controller

--- The screenshots used in the manual page of the DISTRIBUTOR tool where obtained by the distributed generation of DES_SAMPLE.lotos

--- http://cadp.inria.fr/man/svl.html
"controller.bcg" = branching reduction of
CONTROLLER;

-- generation and minimization of the key_path, constrained by the
-- controller generated before

"keypath.bcg" = branching reduction of
KEY_PATH
-[[CTRL_CK, CTRL_SHIFT, CTRL_DK]]
"controller.bcg";

-- generation and minimization of the parallel composition of the
-- minimized controller and the minimized key_path

"controller_keypath.bcg" = branching reduction of
"keypath.bcg"
[[CTRL_CK, CTRL_SHIFT, CTRL_DK]]
"controller.bcg";

-- generation and minimization of the data_path, constrained by the
-- parallel composition of the controller and key_path

"data_path.bcg" = branching reduction of
DATA_PATH
-[[CTRL_CL, CTRL_CR, SUBKEY]]
"controller_keypath.bcg";

-- generation and minimization of the complete DES with abstract bits;
-- gate SUBKEY is left visible for verification purposes

"des.bcg" = branching reduction of
hide CTRL_CL, CTRL_CR, CTRL_CK, CTRL_SHIFT, CTRL_DK in
"data_path.bcg"
[[CTRL_CL, CTRL_CR, SUBKEY]]
"controller_keypath.bcg";

-- Verification of properties

property PROPERTY_1
  "The DES executes indefinitely, i.e., it has no deadlock"
i.
  deadlock of "des.bcg";
  expected FALSE;
end property

property PROPERTY_2
  "The DES can always deliver outputs, and each triplet of inputs is"
"eventually\_followed\_by\_an\_output" 

is "des.bcg" \(\rightarrow\) with evaluator4

library standard.mcl end_library

[ true* ] INEVITABLE ( { OUTPUT ... } );

expected TRUE;

"des.bcg" \(\rightarrow\) with evaluator4

library standard.mcl end_library

macro SEQUENCE (A, B, C) is

(A) .

not ((A) or (B) or (C))*. (B) .

not ((A) or (B) or (C))*. (C)

end_macro

macro PARALLEL (A, B, C) is

SEQUENCE ( (A) , (B) , (C) ) |

SEQUENCE ( (A) , (C) , (B) ) |

SEQUENCE ( (B) , (A) , (C) ) |

SEQUENCE ( (B) , (C) , (A) ) |

SEQUENCE ( (C) , (A) , (B) ) |

SEQUENCE ( (C) , (B) , (A) )

end_macro

[ true* . PARALLEL ( {CRYPT...} , {DATA...} , {KEY...} ) ]

INEVITABLE ( {OUTPUT...} );

expected TRUE;

end property

− − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − − -
To verify the correct function, a prototype implementation is generated from the LNT specification "DES.lnt" (with concrete bits) using the EXEC/CAESAR framework. This prototype is then used to encrypt and
decipher some sample blocks, and the results are compared to the official results.

\% SVL_PRINT_MESSAGE ""
\% SVL_PRINT_MESSAGE "generating DES executable..."

creation of BIT.lnt as a copy of BIT_CONCRETE.lnt
\% sed -e 's/BIT_CONCRETE/BIT/' BIT_CONCRETE.lnt > BIT.lnt

translation from LNT to LOTOS
"DES.lotos" = "DES.lnt";

translation from LOTOS to C
\% caesar.adt -silent DES.lotos 2>&1 | tee -a $SVL_LOG_FILE
\% caesar -silent -exec -e7 DES.lotos 2>&1 | tee -a $SVL_LOG_FILE

compilation of the generated C code
\% $CADP/src/com/cadp_cc -I$CADP/incl -DCAESAR_KERNEL_DELAY=0 -DLNT \%	DES.c gate_functions.c $CADP/src/exec_caeasar/main.c -lm \%	-o des 2>&1 | tee -a $SVL_LOG_FILE

cleanup of generated files
\% SVL_CLEAN_LNT_DEPEND "DES.lnt"
\% SVL_REMOVE "DES.h"
\% SVL_REMOVE "DES.c"
\% SVL_REMOVE "DES.err"
\% SVL_REMOVE "DES.o"
\% SVL_REMOVE "gate_functions.o"
\% SVL_REMOVE "main.o"
\% SVL_RECORD_FOR_CLEAN "des"
\% SVL_RECORD_FOR_CLEAN "input.log"

property PROPERTY_5 (CRYPT, KEY, DATA, OUTPUT)
   "The DES prototype computes the expected result"
is
   \% echo "CRYPT!$CRYPT" > input.log
   \% echo "DATA!$DATA" >> input.log
   \% echo "KEY!$KEY" >> input.log
   \% ./des < input.log
   expected "OUTPUT!$OUTPUT"
end property

check PROPERTY_5 (1, "8000000000000000", "8000000000000000", "6a7fc86c02379a5e");
check PROPERTY_5 (0, "8000000000000000", "6a7fc86c02379a5e");
property PROPERTY_6
"The DES (with concrete bits) correctly computes the encryption" "result of data 0123456789ABCDEF with key 133457799BBCDFF1;" "moreover, when value offers are removed from action labels, the" "LTS generated from the DES with concrete bits is included, modulo" "branching preorder, in the LTS generated from the DES with" "abstract bits"

is
"des_sample.bcg" = branching reduction of
"DES_SAMPLE.Int" : "MAIN_SAMPLE";

"des_sample.bcg" |= with evaluator4
library standard.mccl end_library
[ not ( { OUTPUT ... } )* ] INEVITABLE ( { OUTPUT ... } );
expected TRUE;

branching comparison

  total rename
  "CRYPT.*" -> "CRYPT",
  "DATA.*" -> "DATA",
  "KEY.*" -> "KEY",
  "OUTPUT.*" -> "OUTPUT"
  in "des_sample.bcg"
<=

  hide SUBKEY in

  total rename
  "CRYPT.*" -> "CRYPT",
  "DATA.*" -> "DATA",
  "KEY.*" -> "KEY",
  "OUTPUT.*" -> "OUTPUT"
  in "des.bcg"
expected TRUE;

end property
property PROPERTY_7
    "equivalence between the LNT and LOTOS models"

is
    -- build the LTSs from the LOTOS specifications
    % ( cd LOTOS ; echo "" ; svl )

    -- comparison of the LTSs generated with abstract bits
    branching comparison
        "des.bcg"
    ==
        "LOTOS/des.bcg";
    expected TRUE;

    -- comparison of the LTSs generated with concrete bits
    % if [ -f LOTOS/des.sample.bcg ]
    % then
    branching comparison
        "des_sample.bcg"
    ==
        "LOTOS/des_sample.bcg";
    expected TRUE;
    % fi
end property