Translating the LOTOS NT Data Part into LOTOS Abstract Data Types

David Champelovier, Frédéric Lang

INRIA Rhône-Alpes / VASY
655, avenue de l'Europe
F-38330 Montbonnot Saint Martin

Motivation

- Process algebras (LOTOS, mCRL) are
  - Appropriate to model asynchronous systems formally
  - Equiped with formal verification tools (took years)
- But they are not popular in industry
  - Steep learning curve
  - Lack of trained specifiers
- Need : define new formal description techniques
  - that are more appropriate for an industrial usage, e.g., have an imperative style
  - that enable to reuse existing tools at minimal cost

History

- 1995-1998 : Sighireanu and Garavel define LOTOS NT and participate to the standardization of E-LOTOS (ISO 15437, 2001)
- 2000 : the TRAIAN compiler (from LOTOS NT data part into C) is released and used since then to develop the compilers of VASY (SVL, Exp.Open 2.0, Evaluator 3.0, NTIF, ...)
- 2003 : SENVA launches the SPART initiative, a reflexion on formal description techniques suitable for industrial dissemination
- 2004 : FormalFamePlus Contract between VASY and Bull plans use of LOTOS NT to model critical parts of Bull's next generation high-end servers and the development of a LOTOS NT to LOTOS translator
- 2006 : first release of the LOTOS NT data part to LOTOS translator

This talk

- Short LOTOS reminder
- Presentation of LOTOS NT
- Translation of LOTOS NT data part into LOTOS
- Demo

LOTOS reminder (ISO 8807, 1989)

- Main input formalism of CADP
- Behaviour part inspired from CCS and CSP process algebras
- Data part based on Algebraic Data Types
  - Sets of *sorts* and *operations* defined by conditional algebraic *equations*
  - Operator overloading allowed if signatures differ
- Caesar.adt specific features
  - Priorities between equations
  - Operations split into constructors and non-constructors
  - Oriented equations of the form $E \Rightarrow F\ (P1,\ \ldots,\ Pn) = E'$
  - External C types and operations allowed

LOTOS ADT example

```
type BOOLEAN is
   sorts BOOL
   opns
        FALSE (*! constructor *),
        TRUE (*! constructor *) : -> BOOL
        _and_, _xor_ : BOOL, BOOL -> BOOL
        ...
   eqns
        ofsort BOOL
        forall X, Y : BOOL
        X and TRUE = X;
        X and FALSE = FALSE;
        X = Y => X xor Y = FALSE;
        X xor Y = TRUE;                    (* assuming priority *)
        ...
endtype
```

LOTOS NT

Modules

- Several modules can be defined in separate files
- Modules can import other modules

Example

```
module PLAYER is
        ...
end module
```

PLAYER.lnt

```
module TEAM(PLAYER) is
        ...
end module
```

TEAM.lnt

Types (1/3)

- LOTOS NT allows to define *constructor types*
  - Set of constructors with named typed parameters
  - Enumerated types, records, unions, lists, trees, etc. are special cases
- Examples

```
type WEEKDAY is (* enumerated type *)
        MON, TUE, WED, THU, FRI, SAT, SUN
end type

type DATE is (* record type *)
        DATE (D : NAT, WD : WEEKDAY, M : NAT, Y : NAT)
end type

type NAT_TREE is (* inductive type *)
        LEAF (VAL : NAT),
        NODE (LEFT : NAT_TREE, RIGHT : NAT_TREE)
end type
```

Types (2/3)

- Shorthand support for lists and sets

```
type NAT_LIST is list of NAT end type
```

is a shorthand for

```
type NAT_LIST is
   NIL,
   CONS (HEAD : NAT, TAIL : NAT_LIST)
end type
```

- Standard operations can be defined automatically

```
type NUM is ONE, TWO, THREE with "==", "<=", "<", ">=", ">" end type

type DATE is
   DATE (D : NAT, WD : WEEKDAY, M : NAT, Y : NAT)
   with "get", "set" (* for selectors X.D, ... and updaters X.{D => E}*)
end type
```

Types (3/3)

- Support for external C types and constructors using *pragmas*

```
type INT_32 is
  !external !implementedby "int"
end type

type BYTE is
!implementedby "BYTE"
!printedby "PRINT_BYTE"
  BYTE (B0, B1, B2, B3, B4, B5, B6, B7)
end type
```

- External C code written manually in *MODULE*.f

Functions

- "in" (call by value), "out" and "inout" (call by reference) parameter passing
- Function overloading allowed
- Functions defined using standard algorithmic statements:
  - Local variable declarations and assignments
  - Sequential composition
  - Breakable loops
  - If-then-else conditionals
  - Case statements
  - (Uncatchable) exceptions
- Type checking and variable initialization analysis ensure a clean imperative style

Example (1/3)

```
function COUNT (L : NAT_LIST) : NAT is
   var RESULT : NAT := 0 in
        loop SCAN_L in
                case L in
                var TAIL : NAT in
                  NIL ->
                        break SCAN_L
                | CONS (any NAT, TAIL) ->
                        RESULT := RESULT + 1;
                        L := TAIL
                end case
        end loop;
        return RESULT
   end var
end function
```

Example (2/3)

```
function COUNT (L : NAT_LIST, out EVENS, out ODDS : NAT) : NAT is
   EVENS := 0; ODDS := 0;
   loop SCAN_L in
        case L in
        var HEAD : NAT, TAIL : NAT_LIST in
          NIL ->
                break SCAN_L
        | CONS (HEAD, TAIL) ->
                if IS_EVEN (HEAD) then EVENS := EVENS + 1
                else ODDS := ODDS + 1
                end if;
                L := TAIL
        end case
   end loop;
   return ODDS + EVENS
end function
```

Example (3/3)

```
function GET_HEAD (L : NAT_LIST) : NAT raises EMPTY_LIST : NONE is
   case L in
   var
        HEAD : NAT
   in
     NIL ->
                raise EMPTY_LIST
   | CONS (HEAD, any NAT_LIST) ->
                return HEAD
   end case
end function
```

From LOTOS NT data part into LOTOS ADTs

Translation algorithm

- Based on Ponsini-Fedele-Kounalis-05 (Nice Univ.)
- Extended to handle
  - constructor types
  - case statements
  - reference passing parameters
  - function overloading
  - break statements
  - return statements
  - uncatchable exceptions
- Our extension avoids the generation of useless operations and parameters

Modules and types

LOTOS NT	LOTOS
Module	Type
Type	Sort
Constructor	Constructor operation
Pragma	Pragma
Predefined operation	Operation

Example

```
module TREE is

   type T is
   !printedby "PRINT_TREE"
     LEAF (V : NAT),
     NODE (L, R : T)
     with "get"
   end type

   ...

end module
```

```
type TREE is BOOLEAN, NATURAL, ACTION
   sorts T (*! printedby PRINT_TREE *)
   opns
        LEAF (*! constructor *) : NAT -> T
        NODE (*! constructor *) : T, T -> T
        _NODE_ : T, T -> T
        GET_V : T -> NAT
        GET_L, GET_R : T -> T
   eqns
        ofsort T forall X, Y : T
           X NODE Y = NODE (X, Y);

        ofsort NAT forall V : NAT
           GET_V (LEAF (V) of T) = V;
        ...
endtype
```

Functions

- Each LOTOS NT function is translated into
  - one LOTOS operation for return value (if any)
  - one LOTOS operation for each out/inout parameter
- Examples

F (X : NAT) : NAT	F : NAT -> NAT
F (X : NAT, out Y : BOOL) : NAT	F__2BOOL__NAT : NAT -> NAT
	F__2BOOL__NAT__2 : NAT -> BOOL

- LOTOS operation names contain the signature of out/inout parameters to support overloading

If-then-else statement

- If-then-else is translated into prioritized equations

  Example

```
function F (X : NAT, Y : NAT) : NAT is
   if X > Y then
            return X
   else
            return Y
   end if
end function
```

```
opns
   F : NAT, NAT -> NAT
eqns
ofsort NAT
forall X, Y : NAT
   X > Y => F (X, Y) = X;
   F (X, Y) = Y; (* else *)
```

- Nested if-then-else produce "AND_THEN" conjuncts

  Example

  "if X > 0 then if X < 5 then ..." → "(X > 0) AND_THEN (X < 5) => ..."

Case statements

- Each case statement is assigned a unique number *k* and translates into one operation for each modified variable

  Example

  ```
  case L in
  var TAIL : NAT_LIST in
    NIL -> RESULT := 0
  | CONS (any NAT, TAIL) -> RESULT := 1 + RESULT
  end case
  ```

  →

  ```
  ofsort NAT forall RESULT, ANY__NAT : NAT, TAIL : NAT_LIST
  CASE__k__RESULT (NIL, RESULT) = 0;
  CASE__k__RESULT (CONS (ANY__NAT, TAIL), RESULT) = 1 + RESULT;
  ```

- Symbolic value of RESULT after case :

  CASE__*k*__RESULT (L, RESULT)

Breakable loop statements

- Breakable loop statements are transformed into non-breakable while statements

```
loop L in
  if X = 0 then
        break L
  else
        Y := Y * X
  end if;
  X := X - 1
end loop
```

```
BREAK__1 := FALSE;
while not (BREAK__1) loop
   if X = 0 then
      BREAK__1 := TRUE
   else
      Y := Y * X
   end if;
   if not (BREAK__1) then
      X := X - 1
   end if
end loop
```

while statements

- Each while statement is assigned a unique number *k*
- It translates into 1 function for each modified variable

Example

```
while X > 0 loop
    Y := Y * X;
    X := X - 1
end while
```

→

```
X > 0 => WHILE__k__X (X, Y) = WHILE__k__X (X - 1, Y * X);
WHILE__k__X (X, Y) = X;        (* assuming priority *)
X > 0 => WHILE__k__Y (X, Y) = WHILE__k__Y (X - 1, Y * X);
WHILE__k__Y (X, Y) = Y;        (* assuming priority *)
```

- Value of X after while is WHILE__*k*__X (X, Y)

Return statement

- Single (final) return statement is treated similarly to a variable containing the return value
- Multiple return statements are eliminated (similar to break statements) to get single return

```
if X > 5 then
   return TRUE
end if;
return FALSE
```

→

```
FUNCTION__RETURN := FALSE
if X > 5 then
        FUNCTION__RESULT := TRUE;
        FUNCTION__RETURN := TRUE
end if;
if not (FUNCTION__RETURN) then
        FUNCTION__RESULT := FALSE;
        FUNCTION__RETURN := TRUE
end if;
return FUNCTION__RESULT
```

```
(X > 5) => F (X) = TRUE;
←
F (X) = FALSE;
```

Uncatchable exception

- An enumerated sort EXCEPTION__M containing all exceptions that can be raised in module M is defined
- In function of type T with out/inout parameters V_1:T_1, ..., V_n:T_n, raise X is transformed into

  V_1 := raise__M__T_1(X); V_n := raise__M__T_n(X); return raise__M__T(X)

  (hence the return value and the value of each out/inout parameter after raise has the form raise__M__T_i(X))
- Each raise__M__T_i() is an external function that prints a message and exits

Example

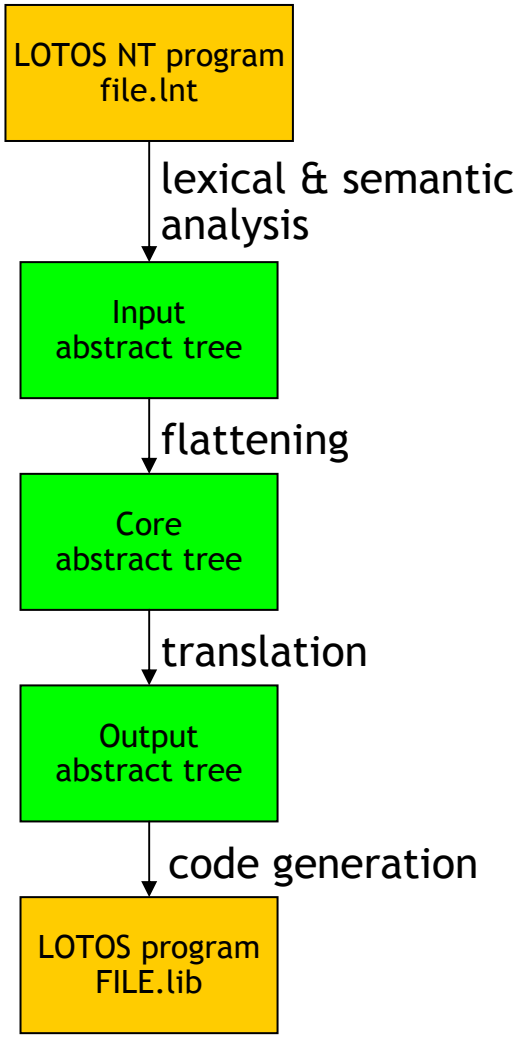
```
function GET_HEAD (L : NAT_LIST) : NAT raises EMPTY_LIST : NONE is
   case L in
   var
        HEAD : NAT
   in
     NIL ->
                raise EMPTY_LIST
   | CONS (HEAD, any NAT_LIST) ->
                return HEAD
   end case
end function
```

→

```
ofsort NAT forall HEAD : NAT, ANY__NAT_LIST : NAT_LIST
GET_HEAD (NIL) = RAISE__M__NAT (EMPTY_LIST);
GET_HEAD (CONS (HEAD, ANY__NAT_LIST)) = HEAD;
```

Lnt2Lotos tool

- Implemented using SYNTAX+TRAIAN
  - 12,800 lines of LOTOS NT
  - 1,100 lines of C
  - 2,100 lines of SYNTAX
- Used by Bull to model critical parts of its next generation high-end servers (Novascale)
  - 2290 LOTOS NT lines → 1661 LOTOS lines (-27,5 %)
- Validated on 63 tests
  - 6210 LOTOS NT lines → 4914 LOTOS lines (-21 %)
- Pen & paper proof of the translation



Future work

- Provide more types (e.g., arrays)
- Define an interface language for separate type checking
- Generate code for other tools
  - mCRL
  - ADT based theorem prover
- Implement a full LOTOS NT to LOTOS translator (including behaviours)