NAME

xtl – language for value-based temporal logic formulas

DESCRIPTION

XTL (eXecutable Temporal Language) is a functional programming language interpreted over LTSs (Labelled Transition Systems) encoded in the BCG (Binary Coded Graph) file format.

XTL can typically be used for implementing temporal logic operators, by describing their fixed point semantics as iterative or recursive computations over sets of states.

More generally, XTL enables one to perform any computation on a BCG graph: for instance, it can compute the branching factor of a graph, print its list of labels, etc.

XTL programs are compiled and evaluated on BCG graphs by using the **xtl**(LOCAL) model checker.

LTS MODEL

An XTL program is evaluated over an LTS model generated from a source program to be verified. This LTS, encoded in a BCG file, contains the following elements:

- a set of *states* of the source program. A state is represented as a tuple containing the values of all the variables of the program (the so-called "state vector").

- a set of *labels* performed by the program. An label is represented as a list of typed values. In BCG files generated from LOTOS programs, the labels have the form *G v1 ... vm* (with m >= 0), where *G* is a gate name.

- a *transition relation* between the states of the program. A transition, represented as a triple (*s1*, *a*, *s2*), means that the program can move from state *s1* to state *s2* by performing a transition labelled with *a*.

- the *initial state* of the program.

SYNTAX AND SEMANTICS OF XTL

The syntax of each XTL construct is defined by a BNF grammar and the semantics is described informally. Terminal symbols are enclosed in double quotes. Optional elements are enclosed in square brackets. Sus-

pension points are used to denote zero or more repetitions of an element. The meaning of grammar symbols is given in the table below. The axiom of the grammar is the non-terminal symbol *PG*.

Symbol	Description
K	constant
x	variable
T	simple type
G	gate
F	function
M	macro
P	macro parameter
E	expression
O	offer
D	variable declaration
RT	result type
OP	operator
FD	function definition
MD	macro definition
LI	library inclusion
ETD	external type declaration
EFD	external function declaration
EID	external include directive
ELD	external compile directive
PG	program

LEXICAL ELEMENTS

The lexical units of XTL are: *keywords*, *identifiers* and *separators*. The keywords, listed below, must be written in lower-case letters.

any	**end_def**	**exists**	**of**
apply	**end_exists**	**for**	**on**
assert	**end_for**	**forall**	**then**
def	**end_forall**	**from**	**to**
else	**end_if**	**if**	**use**
else_if	**end_let**	**in**	**where**
end_assert	**end_use**	**let**	

The identifiers are grouped into two classes:

- *external identifiers* denote the objects defined in the source program to be verified (these objects are contained in the BCG file). These identifiers may be written like the internal ones, as long as there is no clash between them and the normal identifiers or the XTL keywords. In the latter case, external identifiers must be preceded by a dollar character (e.g., **$let**) to avoid clashes.

- *internal identifiers* denote the objects defined in the XTL program (or predefined in the XTL language). These identifiers may be either *normal*, i.e., built from letters, digits and underscores (beginning with a letter or an underscore), or *special*, i.e., built from the characters #, %, &, *, +, -, ., /, >, =, <, @, \, ^, and ~. In normal identifiers, no difference is made between lower- and upper-case letters. The special identifiers are useful for mathematical and/or logical notations (e.g., '+', '/\', etc.).

The separators are sequences containing space, tab, and newline characters, as well as *comments*, enclosed between **(*** and ***)**.

INTERNAL AND EXTERNAL TYPES

XTL is a strongly-typed language: every object used in an XTL program must have a unique type, which is statically determined. XTL does not provide a mechanism for type definition, although anonymous tuple types (see below) can be implicitly created. The types allowed in XTL fall into the following classes:

- *external types* are imported from the source program to be verified. These types are contained in the type area of the BCG file and can be referenced in XTL programs using external identifiers. Other externally defined types can be included in the XTL programs by means of special directives (see DIRECTIVES below).
- *internal types* are either predefined in the language, or defined by the user.

INTERNAL AND EXTERNAL FUNCTIONS

The functions (also called *operators*) that can be used in an XTL program fall into the following classes:

- *external functions* are defined in the source program to be verified. They are contained in the function area of the BCG file and can be referenced in XTL programs using external identifiers.

- *internal functions* are either predefined in XTL (a list of the predefined functions has been given in the above paragraphs), or user-defined (see FUNCTIONS below for the syntax of function definitions).

In XTL, functions can be *overloaded*, i.e., have the same identifier but different types for their arguments and/or result. The XTL compiler resolves overloading statically (i.e., at compile-time) or issues an error message if the context does not permit to resolve overloading (in such case, the "**of**" operator can be used to supply typing information).

In XTL, a function *F* can be either *prefixed*, i.e., its calls are written "*F* (*E1*, ..., *En*)", or *infixed*, i.e., *F* has two arguments and its calls are written "*E1 F E2*".

Most predefined functions are infixed, except "**extract**" and "**replace**". For user-defined functions, the prefix or infix nature is specified when the function is declared.

Also, unary minus operators must be written in functional notation, i.e., **-(1)** instead of **-1**, **-(X)** instead of **-X**, etc.

BOOLEAN TYPE AND FUNCTIONS

The Boolean type is named **boolean**. The Boolean constants **true** and **false** are defined as 0-ary functions. The table below gives the predefined functions for this type:

Operator	Meaning
true, false : -> boolean	boolean constants
not : boolean -> boolean	negation
or, and, implies,	connectors and
=, <> : boolean, boolean -> boolean	relational operators
replace : boolean, boolean -> boolean	replacement operator

The "**replace**" operator (which is overloaded for all types) takes two arguments of the same type and returns the second one. It is useful for use within the "**for**" expressions (see EXPRESSIONS below).

INTEGER TYPE AND FUNCTIONS

The signed integer type is named **integer**. Integer constants are noted with a C-like syntax and should be preceded of a sign "**+**" or "**-**", e.g., **-123**, **+123**, **+0**, etc. The table below gives the predefined functions for this type:

Operator	Meaning
-, + : integer, integer -> integer	binary minus, plus
- : integer -> integer	unary minus
*, div : integer, integer -> integer	multiplication, division
mod : integer, integer -> integer	modulo
** : integer, integer -> integer	power
integer : real -> integer	real to integer
integer : character -> integer	character to integer
<, <=, >, >=, =, <> : integer, integer -> boolean	relational operators
replace : integer, integer -> integer	replacement operator

NATURAL TYPE AND FUNCTIONS

The unsigned integer type is named **natural**. Natural constants are noted with a C-like syntax and should never be preceded of a sign, e.g., **123**, **0**, etc. The table below gives the predefined functions for this type:

Operator	Meaning
-, + : natural, natural -> natural	binary minus, plus
*, div : natural, natural -> natural	multiplication, division
mod : natural, natural -> natural	modulo
** : natural, natural -> natural	power
natural : real -> natural	real to integer
natural : character -> natural	character to natural
<, <=, >, >=, =, <> : natural, natural -> boolean	relational operators
replace : natural, natural -> natural	replacement operator

By default, an unsigned number **123** denotes either a value of type **natural** or a value of type **integer**. Such overloading can be resolved explicitly by the user (using an "**of**" operator) or implicitly by the context (for instance, in **(X + 123)**, the type of **X** will determine the type of **123**). If the context does not permit to resolve overloading, then the constant will ultimately be given the type Natural. To summarize:
- **+123** has the **integer** type.
- **-123** has the **integer** type.
- **123 of integer** has the **integer** type.
- **123 of natural** has the **natural** type.
- **123** has either the **natural** or **integer** type (with a final priority for **natural**).

REAL TYPE AND FUNCTIONS

The floating-point number type is named **real**. Real constants are noted with a C-like syntax, e.g., **3.1416**, **-9.98E-6**, etc. The table below gives the predefined functions for this type:

Operator	Meaning
-, + : real, real -> real	binary minus, plus
- : real -> real	unary minus
*, / : real, real -> real	multiplication, division
** : real, real -> real	power
real : integer -> real	integer to real
<, <=, >, >=, =, <> : real, real -> boolean	relational operators
replace : real, real -> real	replacement operator

CHARACTER TYPE AND FUNCTIONS

The character type is named **character**. Character constants are noted with a C-like syntax, e.g., **'a'**, **'\n'**, **'\012'**, **'\x1A'**, etc. The table below gives the predefined functions for this type:

Operator	Meaning
-, + : character, integer -> character	subtract, add integer
character : integer -> character	integer to character
<, <=, >, >=, =, <> : character, character -> boolean	relational operators
replace : character, character -> character	replacement operator

STRING TYPE AND FUNCTIONS

The character-string type is named **string**. String constants are noted with a C-like syntax, e.g., **"abc"**, **"123"**, **"Hello\n"**, etc. The table below gives the predefined functions for this type:

Operator	Meaning
null : -> string	empty string
length : string -> integer	string length
extract : string, integer -> character	character extraction
=, <> : string, string -> boolean	relational operators
replace : string, string -> string	replacement operator

RAW TYPE AND FUNCTIONS

The **raw** type represents a byte string that denotes a value of unknown type, i.e., whose type is not one of the other predefined types. Raw constants are noted with a C-like syntax but using backquotes rather than double quotes, e.g., **`foo`**, **`CONS (0, NIL)`**, or **`{0, 1}`**, etc. The table below gives the predefined functions for this type:

Operator	Meaning
null : -> raw	empty raw data
length : raw -> integer	raw data length
extract : raw, integer -> character	character extraction
=, <> : raw, raw -> boolean	relational operators
replace : raw, raw -> raw	replacement operator

By default, the string notation **"..."** denotes either a value of type **string** or a value of type **raw**. Such overloading can be resolved explicitly by the user (using an "**of**" operator) or implicitly by the context (for instance, in **(X = "foo")**, the type of **X** will determine the type of **"abc"**). If the context does not permit to resolve overloading, then the constant will ultimately be given the type String. To summarize:
- **`foo`** has the **raw** type.
- **"foo" of string** has the **string** type.
- **"foo" of raw** has the **raw** type.
- **"foo"** has either the **string** or **raw** type (with a final priority for **string**).

META-TYPES AND FUNCTIONS

The types **state**, **label**, **edge**, **stateset**, **labelset**, and **edgeset** (also called *meta-types* because they refer to the LTS model rather than to the objects defined in the source program to be verified) denote the states, labels, edges, sets of states, sets of labels, and sets of edges of the LTS, respectively.

The table below gives the predefined operators over these types ("**set**" denoting either **stateset**, **labelset**, or **edgeset**). These operators allow to explore (forward and/or backward) the transition relation of the LTS, to combine sets of states, labels and edges, and also to obtain information about the LTS.

Operator	Meaning
init : -> state	initial state
succ : state -> stateset	successor states
pred : state -> stateset	predecessor states
out : state -> edgeset	successor transitions
in : state -> edgeset	predecessor transitions

=, <> : state, state -> boolean	comparison operators
replace : state, state -> state	replacement operator
visible : label -> boolean	visibility test
string : label -> string	label to string
=, <> : label, label -> boolean	comparison operators
replace : label, label -> label	replacement operator
source : edge -> state	transition source state
target : edge -> state	transition target state
label : edge -> label	transition label
succ : edge -> edgeset	successor transitions
pred : edge -> edgeset	predecessor transitions
=, <> : edge, edge -> boolean	comparison operators
replace : edge, edge -> edge	replacement operator
empty, false : -> set	empty set
full, true : -> set	full set
comp, not : set -> set	complementation
union, or : set, set -> set	union
inter, and : set, set -> set	intersection
implies : set, set -> set	implication
iff : set, set -> set	equivalence
diff : set, set -> set	difference
includes : set, set -> boolean	inclusion test
insert, remove : set, elem -> set	insertion, removal
among : elem, set -> boolean	membership
card : set -> number	cardinal
=, <> : set, set -> boolean	comparison operators
replace : set, set -> set	replacement operator

For convenience, some of the operators above are overloaded (see FUNCTIONS below). Also, some of the set operators have synonyms (e.g., **union** has the synonym **or**) allowing a more intuitive writing of boolean properties.

NUMBER TYPE AND FUNCTIONS

The BCG file format associates a unique number to every state, label, and edge of the LTS. The XTL language provides the **number** type for the manipulation of state, label, and edge numbers. The table below gives the predefined operators of type **number**.

Operator	Meaning
number_of_states : -> number	number of states
number_of_labels : -> number	number of labels
number_of_edges : -> number	number of edges
number : state -> number	state number
number : label -> number	label number
number : edge -> number	edge number
-, + : number, number -> number	binary minus, plus
*, div : number, number -> number	multiplication, division

mod : number, number -> number	modulo
** : number, number -> number	power
number : character -> number	conversion from/to character
character : number -> character	
number : integer -> number	conversion from/to integer
integer : number -> integer	
number : real -> number	conversion from/to real
real : number -> real	
<, <=, >, >=, =, <> : number, number -> boolean	relational operators
replace : number, number -> number	replacement operator

All arithmetic operators (except unary minus) and relational operators of type **integer** are also available for type **number**. Moreover, the values belonging to these two types can be freely mixed within arithmetic and relational expressions. The main difference between these two types concerns the manipulation of large numbers: the maximum value of type **integer** is **2^31-1** = **2,147,483,647** on 32-bit machines and **2^63-1** = **9,223,372,036,854,775,807** on 64-bit machines, whereas the maximum value of type **number** is **2^32-1** = **4,294,967,295** on 32-bit machines and **2^64-1** = **18,446,744,073,709,551,615** on 64-bit machines. Therefore, for large LTSs, any information concerning the LTS structure (e.g., cardinality of state/edge/label subsets, depth/breadth of the LTS, number of breadth-first levels, etc.) should be manipulated using values of type **number** instead of type **integer** in order to avoid overflows.

Conversion operators between the **number** type and the types **character**, **integer**, and **real** are also provided. Constants of type **number** are written in decimal notation preceded by a **'#'** character (e.g., **#4294967295**). An alternative way of representing constants of type **number** that fit in the range of type **integer** is by means of the conversion operator from **integer** to **number** (e.g., the values **number(2147483647)** and **#2147483647** are the same).

ACTION TYPE AND FUNCTIONS

The XTL language has an action type associated to XTL expressions that return no value but perform side effects. These are used to print data values on the POSIX standard output stream (**stdout**). The table below gives the predefined operators of type **action** (T may be any type).

The **print** and **printf** operators behave differently on strings: **print** prints the string enclosed between double quotes and converts unprintable characters and escape sequences to three-digit octal notation, whereas **printf** does not add double quotes and interprets execute sequences. For instance, **print ("Hello!\n")** displays **"Hello!\012"**, whereas **printf ("Hello!\n")** displays **Hello!** followed by a line-feed. So, **print** should be used to display string values (e.g., label offers of the string type), whereas **printf** should be used to display messages.

Operator	Meaning
nop : -> action	inaction
fby : action, action -> action	sequential composition
print : T -> action	value printing
printf : string -> action	string formatted printing

ANONYMOUS TUPLE TYPES

These types denote structures containing fields of different types. The fields can be of any type (including anonymous tuples, which enables nesting of tuples). These types are noted "(" *T0*"," ..."," *Tn* ")" and are essentially used in situations when several values must be computed and returned simultaneously. The equivalence of tuple types is defined structurally. The tuple values are noted "(" *E0*"," ..."," *En* ")" (see EXPRESSIONS below).

EXPRESSIONS

The expressions allowed in XTL use the following auxiliary constructs.

Offers An *offer* is a construct allowing to match a value contained in a transition label. The offers have the following syntax:

O ::= "?" *x* ":" *T*

| "!" *E*

| "_"

An offer "?" *x* ":" *T* matches a value *v* iff *v* has the type *T*; in case of matching, the variable *x* is also assigned the value *v*. An offer "!" *E* matches a value *v* iff the expression *E* evaluates to *v*. An offer "_" (wildcard) matches a value *v* of any type.

Result types

The result types denote either simple types, or anonymous tuple types. They have the following syntax:

RT ::= *T*

| "(" *RT0*"," ..."," *RTn* ")"

These types may occur in variable declarations (see the next paragraph) as well as in function declarations (see FUNCTIONS below).

Variable declarations

The variable declarations have the following syntax:

D ::= *x* ":" *RT*

| "any" *RT*

| "(" *D0*"," ..."," *Dn* ")"

A declaration *x* ":" *RT* defines the variable *x* of type *RT*. A declaration "any" *RT*, which is meaningful only inside a declaration of anonymous tuple type, defines a placeholder of type *RT* standing for a field of the tuple. Declarations of the form "any" *RT* can occur only in the "let" expressions (see below). A declaration "(" *D0*"," ..."," *Dn* ")" defines a tuple whose fields are denoted by the variables and placeholders occurring inside *D0*, ..., *Dn*.

Operators

The operators occur in the "**for**" expressions (see below); they denote calls of binary functions on two arguments. They have the following syntax:

OP ::= *F*

| "(" *OP0* "," "..." "," *OPn* ")"

An *F* operator denotes a call of the binary function *F*. An "(" *OP0*"," ..."," *OPn* ")" operator denotes a call of a binary function on two arguments of tuple type; the result is a tuple whose fields are the results of the calls of *OP0*, ..., *OPn* on the corresponding fields of the arguments. There is a static semantics constraint on the operators *OP*: their left arguments must have the same types as their results.

The syntax of expressions is given by the following grammar:

```
E ::= K

    | F "(" E0"," ..."," En ")"

    | E0 "of" RT

    | E0 "->" "[" O0 ... On [ "..." ] [ "where" E1 ] "]"

    | "{" E0"," ..."," En "}"

    | "(" E0"," ..."," En ")"

    | "let" D0 "=" E0"," ..."," Dn "=" En "in"
          En+1
      "end_let"

    | "if" E0 "then" E'0
          "else_if" E1 "then" E'1
          ...
          "else_if" En "then" E'n
          "else" E'n+1
      "end_if"

    | "assert" E0"," ..."," En "in"
          En+1
      "end_assert"

    | "use" x0"," ..."," xn "in"
          E
      "end_use"

    | "for" [ x0 ":" T0 [ "among" E0 ]"," ..","
              xn ":" Tn [ "among" En ] ]
          [ "in" D ]
          [ "while" E'1 ]
          [ "where" E'2 ]
          "apply" OP
          "from" E'3
          "to" E'4
      "end_for"
```

```
|    "forall" x0 ":" T0 [ "among" E0 ]"," ..."," 
                xn ":" Tn [ "among" En ]
     "in"
         En+1
     "end_forall"

|    "exists" x0 ":" T0 [ "among" E0 ]"," ..."," 
                xn ":" Tn [ "among" En ]
     "in"
         En+1
     "end_exists"

|    "<|" OP "on" x0 ":" T0 [ "among" E0 ]"," ..."," 
                    xn ":" Tn [ "among" En ]
          [ "where" E' ]
     "|>" En+1

|    "{" x ":" T [ "among" E0 ] "where" E1 "}"
```

The semantics of expressions is described informally below.

K

is a literal constant of a predefined type (see TYPES, FUNCTIONS AND CONSTANTS above).

F "(" *E0*"," ..."," *En* ")"

denotes a call of the function *F* on the arguments *E0*, ..., *En*. The arguments must be compatible (in number and types) with the formal parameters given in the definition of *F* (see FUNCTIONS below).

E0 "of" *RT*

specifies that expression *E0* has type *RT*. This mechanism allows to solve ambiguities that may be caused by function overloading (see FUNCTIONS below).

E0 "->" "[" *O0* ... *On* ["..."] ["where" *E1*] "]"

is an expression of type **boolean**, called matching expression. *E0* must be of type **label** or **edge**. The construct between "**[**" and "**]**" is called an label pattern: it allows to match the content of the label denoted by *E0* (if *E0* is of type **label**) or by **label** (*E0*) (if *E0* is of type **edge**), possibly extracting the values of its different fields. The optional construct "..." matches a sequence of zero or more values of any type.

For convenience, in the verification of LOTOS programs, the first offer *O0* (intended to match a gate) can be simply written *G* (i.e., without the "**!**" sign).

The variables contained in *O0*, ..., *On* (if any) are visible in the optional expression *E1*, which must be of type **boolean**.

The XTL compiler examines the labels present in the BCG graph to check if there exist labels that correspond, in the number of fields and types of their fields, to the list of offers *O0*, ..., *On*. If no such label is found, then the compiler issues a warning message (this is a form of *vacuity checking*).

The compiler also uses the types of field labels to resolve overloading ambiguities (if any) in offers. For instance, the offer **1** in the label pattern "**G !1**" will be considered of **integer** if only integer values are passed on gate **G** in the BCG graph on which the XTL program is

evaluated.

A matching expression matches a label *G v1 ... vm* iff: m = n if "..." is absent, or m >= n otherwise; *O0* matches *G*; *O1*, ..., *On* match *v1*, ..., *vn*; *E1*, if present, evaluates to **true** in the context of the variables assigned in the offers. A matching expression evaluates to **true** in case of successful matching of a label (in this case, all the variables contained in the offers have been initialized) and to **false** otherwise.

```
"{" E0"," ..."," En "}"
```

denotes a value of type set defined explicitly (i.e., by enumerating all its elements). The expressions *E0*, ..., *En* must all be of type **stateset**, **labelset**, or **edgeset**.

```
"(" E0"," ..."," En ")"
```

denotes a value of type tuple, containing n fields whose values are given by *E0*, ..., *En*.

```
"let" D0 "=" E0"," ..."," Dn "=" En "in"
    En+1
"end_let"
```

allows to declare and initialize variables. The types of the initialization expressions *E0*, ..., *En* must be compatible with the types corresponding to the declarations *D0*, ..., *Dn*. All the variables occurring in the declarations *Di* are initialized with the values of *Ei*, and are visible in the expression *En+1*. The "**let**" expression also allows to extract the fields of tuple values and to assign them to variables.

```
"if" E0 "then" E'0
    "else_if" E1 "then" E'1
     ...
    "else_if" En "then" E'n
    "else" E'n+1
"end_if"
```

allows the conditional evaluation of expressions. The expressions *E0*, ..., *En* must be of type **boolean**, whereas *E'0*, ..., *E'n+1* must be of the same type (which is also the type of the "if" expression). The evaluation of an "if" expression proceeds as follows: the conditions *Ei* are evaluated (for 1 <= i <= n) and the value of the "if" expression is equal to the value of the first *E'i* for which *Ei* is true. If none of the conditions evaluates to true, the resulting value is given by *E'n+1*. If a condition *Ei* is a matching operator, the variables assigned by the offers inside *Ei* are visible only in the corresponding expression *E'i*.

```
"assert" E0"," ..."," En "in"
    En+1
"end_assert"
```

allows to stop the execution of an XTL program if a given condition does not hold. The expressions *E0*, ..., *En* must be of type **boolean**. If all the assertions evaluate to **true**, the value of the "assert" expression is equal to the value of *En+1*; otherwise, the execution of the XTL program is interrupted and an error message is displayed on the POSIX standard output stream (**stdout**).

```
"use" x0"," ..."," xn "in"
    E
"end_use"
```

uses the values of variables *x0*, ..., *xn* without changing their value and then evaluates the expression *E*. The "use" expression allows to write XTL programs in which every variable defined is used, which avoids the warnings of the form "variable set but not used" issued during compilation of the C code generated by XTL. Although the usage of the "use" expression is harmless and does not impact in any way the evaluation of XTL expressions, it should be employed only in the (rare) situations where it is really needed.

```
"for" [ x0 ":" T0 [ "among" E0 ]"," ..."," 
        xn ":" Tn [ "among" En ] ]
    [ "in" D ]
    [ "while" E'1 ]
    [ "where" E'2 ]
    "apply" OP
    "from" E'3
    "to" E'4
"end_for"
```

allows the iterative evaluation of expressions. The *iteration variables x0*, ..., *xn* range over the iteration domains *T0* ["among" *E0*], ..., *Tn* ["among" *En*], respectively. The expressions *Ei* must be of type "**set of** *Ti*": thus, iterations over sets of states, labels, or edges are allowed. There is an exception concerning integer numbers and characters: for these types, domains of the form "{" *E1* "..." *E2* "}", meaning all the integers (resp. characters) between *E1* and *E2*, are also allowed. The variables declared in the "in" *D* declaration (if present) are called *accumulators*. Both iteration variables and accumulators are visible in the expressions *E'1*, *E'2* (if present), and *E'4*, but not in *E'3*. The expressions *E'1* and *E'2* must be of type **boolean**.

Assuming that all the optional constructs are present, the evaluation of a "**for**" expression proceeds as follows. The accumulators are initialized to the value of *E'3*. Let us note *v* the current values of the accumulators (*v* may be of type tuple). Then, for each value of the iteration variables in their corresponding domains, the expression *v OP E'4* is evaluated and its result is stored in the accumulators. The value of the "**for**" expression is equal to the value of the accumulators obtained after the last iteration. The "where" clause allows to perform only those iterations for which the expression *E'2* evaluates to **true**. The "while" clause allows to stop the iterations when the expression *E'1* becomes **false**.

Particular cases of "**for**" expressions may be obtained by eliminating some (or all) of the optional constructs. If the iteration variables are absent, the "**for**" becomes an infinite loop: in this case, the "**while**" clause must be present in order to stop the evaluation. If the "**in**" declaration is absent, the accumulator is implicit: it is used internally to store the results of intermediate iterations, but it cannot be used in *E'1*, *E'2*, nor *E'4*.

The "**for**" expressions are useful for describing iterative computations. For instance, the expression below displays on the standard output the sequence of Fibonacci numbers smaller than 1000:

```
for
     in    (xn_plus_1, xn: integer, a:action)
    while xn < 1000
    apply (replace, replace, fby)
    from  (1, 1, nop)
    to    (xn_plus_1 + xn, xn_plus_1,
           print (xn) fby printf ("\n"))
end_for
```

```
"forall" x0 ":" T0 [ "among" E0 ]"," ..."," 
         xn ":" Tn [ "among" En ]
"in"
    En+1
"end_forall"
```

is the universal quantifier. It is equivalent to the following "**for**" expression:

```
"for" x0 ":" T0 [ "among" E0 ]"," ..."," 
      xn ":" Tn [ "among" En ]
    "apply" and
    "from"  true
    "to"    En+1
"end_for"
```

The type of *En+1* may be either **boolean**, **stateset**, **labelset**, or **edgeset**. In the three latter cases, the **true** and **and** operators correspond to full set and set intersection, respectively (see TYPES, FUNCTIONS AND CONSTANTS above).

```
"exists" x0 ":" T0 [ "among" E0 ]"," ..."," 
         xn ":" Tn [ "among" En ]
"in"
    En+1
"end_exists"
```

is the existential quantifier. It is equivalent to the following "**for**" expression:

```
"for" x0 ":" T0 [ "among" E0 ]"," ..."," 
      xn ":" Tn [ "among" En ]
    "apply" or
    "from"  false
    "to"    En+1
"end_for"
```

The type of *En+1* may be either **boolean**, **stateset**, **labelset**, or **edgeset**. In the three latter cases, the **false** and **or** operators correspond to empty set and set union, respectively (see TYPES, FUNCTIONS AND CONSTANTS above).

```
"<|" OP "on" x0 ":" T0 [ "among" E0 ]"," ..."," 
             xn ":" Tn [ "among" En ]
     [ "where" E' ]
"|>" En+1
```

is an abbreviated form of the "**for**" expression, called *iterator*. It is equivalent to the following "**for**" expression:

```
"for" x0 ":" T0 [ "among" E0 ]"," ..."," 
      xn ":" Tn [ "among" En ]
    [ "where" E' ]
    "apply" OP
    "from"  init_OP
    "to"    En+1
"end_for"
```

Here *init_OP* is a "start" value associated by default to the operator *OP*. Only the binary prede-

fined operators that have initial values assigned by default can be used in the iterators. These operators, together with their initial values, are given in the table below (where "*set*" denotes either **stateset**, **labelset**, or **edgeset**).

Operator *OP*	*init_OP*
or : boolean, boolean -> boolean	false
and : boolean, boolean -> boolean	true
+ : integer, integer -> integer	0
* : integer, integer -> integer	1
union, or : set, set -> set	empty, false
inter, and : set, set -> set	full, true
diff : set, set -> set	full
insert : set, elem -> set	empty
remove : set, elem -> set	full
fby : action, action -> action	nop

The iterators allow to express iterative computations in a form close to the mathematical notation. For example, the sum of the integer numbers from 1 to 100 may be computed with the following iterator:

```
<| + on K:integer among { 1 ... 100 } |> K
```

Similarly, the number of transitions labelled with "SEND" labels may be computed as follows:

```
<| + on T:edge where T -> [ SEND ... ] |> 1
```

"{" *x* ":" *T* ["among" *E0*] "where" *E1* "}"

is a set value defined by specifying a predicate *E1* characterizing the elements of the set. This construct is also a particular case of "**for**" expression:

```
"for" x ":" T [ "among" E0 ]
    "where" E1
    "apply" union
    "from"  empty
    "to"    "{" x "}"
"end_for"
```

The implicitly defined sets allow to compute sets of states, labels or transitions in a form close to the mathematical notation. For example, the set of deadlock states (i.e., states having no successors) can be computed by the XTL expression below:

```
{ S:state where succ (S) = empty }
```

FUNCTIONS

The XTL language allows to define and use functions. The syntax of function definitions is given by the grammar below:

```
FD      ::=  "def" F "(" x1 ":" T1"," ..."," xn ":" Tn ")" ":" RT "="
                 [ FD1 ... FDm ]
                 E
                 [ "where" FDm+1 ... FDm+p ]
             "end_def"

       |     "def" "_" F "_" "(" x1 ":" T1"," x2 ":" T2 ")" ":" RT "="
                 [ FD1 ... FDm ]
                 E
                 [ "where" FDm+1 ... FDm+p ]
             "end_def"
```

The first construct above defines a function *F* having the parameters *x1*, ..., *xn* of types *T1*, ..., *Tn*, the body *E*, and returning a result of type *RT*. The calls of *F* have the form:

```
F "(" E1"," ..."," En ")"
```

where the arguments *E1*, ..., *En* must be compatible (in number and types) with the parameters of *F*. The result of the call is equal to the value of *E* computed in a context in which the parameters *xi* are assigned the values of *Ei* for all i between 1 and n. The body of *F* may also contain definitions of local functions, placed either before *E*, or after *E* and the keyword "**where**". These functions are visible only in the expression *E*. Recursive functions are allowed. Also, functions can be overloaded: several functions with the same name, but different profiles, may be defined in the same scope. In case of ambiguity of the type of a function call, the "**of**" expression may be used to precise a unique type (see EXPRESSIONS above).

The second construct above defines a binary infixed function *F* having the parameters *x1*, *x2* of types *T1*, *T2*, and returning a result of type *RT*. The function *F* behaves like an ordinary binary function, except that its calls may be written either in the form *F* "(" *E1*"," *E2* ")", or in the form *E1 F E2*. This kind of functions allow to write expressions involving infixed operators using a syntax close to the mathematical notation. For example, the predefined function "**+**" over integers is in fact an infixed operator: the user may write either + (1, 2), or 1 + 2. The infixed operators (both predefined and user-defined) are right-associative: e.g., the expression 1 + 2 + 3 is parsed as 1 + (2 + 3).

Functions can be, of course, recursive (either directly, or transitively). For example, the recursive function below computes the factorial of an integer number:

```
def fact (n:integer) : integer =
    assert n >= 0 in
        if n = 0 then
            1
        else
            n * fact (n - 1)
        end_if
    end_assert
end_def
```

This function also checks whether its argument is greater or equal to 0 using an "**assert**" expression; if this is not the case, the program execution is aborted with an appropriate error message.

MACROS

The XTL compiler allows to define and use macros. The syntax of macro-definitions is given by the following grammar:

```
MD      ::=  "macro" M "(" P1 "," ... "," Pn ")" "="
                 <text>
             "end_macro"
```

The above construct defines a macro *M* having the parameters *P1*, ..., *Pn* and the body <text>, which is a portion of text built using the characters allowed by the XTL language (see LEXICAL ELEMENTS above). The calls of *M* have the form:

```
M "(" <text1>"," ..."," <textn> ")"
```

where the arguments <text1>, ..., <textn> are portions of XTL text. The result of the call is <text> in which all the occurrences of the parameters *Pi* have been syntactically substituted with the arguments <texti>, for all i between 1 and n. The following syntactic restriction must be satisfied: the keywords contained in each <texti> argument must be well-bracketed, i.e., if <texti> contains a keyword opening an expression (e.g., "**for**"), then <texti> must also contain the corresponding keyword closing the expression (e.g., "**end_for**"). This also applies to the non-alphabetic keywords such as "**(**", "**)**", "**[**", "**]**", "**{**", "**}**". A macro is visible from the point of its definition until the end of the XTL program. The macros may be overloaded: several macros with the same name, but different arities, may be defined in the same scope.

LIBRARIES

There is also possible to include in the text of an XTL program external libraries, typically containing definitions of temporal operators. The inclusion command has the following syntax:

```
LI      ::=  "library" <file0.xtl>"," ..."," <filen.xtl>
             "end_library"
```

At the compilation of the program, the above construct is syntactically replaced with the contents of the files <file0.xtl>, ..., <filen.xtl>, placed one after the other in this order. The XTL compiler searches the included files first in the current directory, then in the directory referenced by **$CADP/src/xtl**. Multiple inclusions of the same file are silently discarded, unless the **-warning** option is passed to the compiler; in this case, appropriate messages are issued.

DIRECTIVES

The XTL compiler allows to declare and use in an XTL program data types and functions implemented externally in C. The directive for an external type declaration has the following syntax:

```
ETD  ::=  "type" T
                [ "!" "implementedby" "<C_type>" ]
                [ "!" "comparedby" "<C_compare>" ]
                [ "!" "enumeratedby" "<C_iterate>" ]
                [ "!" "printedby" "<C_print>" ]
          "end_type"
```

where *T* is the identifier of the declared type and the optional pragmas preceded by "!" refer to the C implementation of *T*: <C_type> is the identifier of the C type implementing *T*; <C_compare> is an operator for comparing two values of type *T*; <C_iterate> is a macro allowing to iterate over all values of type *T*; and <C_print> is a printing function for values of type *T*. The C names declared by these pragmas cannot be used directly in the XTL program, but are used in the C code generated by the XTL compiler.

The directive for an external function declaration has the following syntax:

```
EFD  ::=  "func" F "(" T1"," ..."," Tn ")" ":" RT
                [ "!" "implementedby" "<C_func>" ]
          "end_func"
```

where *F* is the identifier of the declared function, *T1*, ..., *Tn* are the types of its parameters, and *RT* is the type of its result. The optional pragma "implementedby" declares the name <C_func> of the C function implementing *F*. This C identifier cannot be used in the XTL program, but is used in the C code generated by the XTL compiler.

The external types and functions declared using the directives above may be implemented in (one or more) C files file1.c, ..., filen.c that must be included in the C code generated by the XTL compiler. This is done using the following external include directive:

```
EID  ::=  "include" "<file1>", ..., "<filen>" "end_include"
```

where <file1>, ..., <filen> are the names of the C source files (".c" and/or ".h") that must be included. This directive is translated into corresponding "#include" pre-processor commands in the C code generated by the XTL compiler.

The compilation and linkediting of the C modules included may require specific parameters. These can be indicated to the XTL compiler using the following external compile directive:

```
ELD  ::=  "flag" "<C_compiler_directives>" "end_flag"
```

where <C_compiler_directives> is a portion of command-line for invoking the C compiler (typically containing "-I", "-L", and "-l" options) that specifies the desired compilation parameters. This portion of command-line is used by the XTL compiler when creating the binary file of the generated C code.

As an example, the XTL program below uses the CAESAR_ERROR() function for error handling, which is declared externally in the "caesar_standard.h" file and defined in the OPEN/CAESAR library libcaesar.a. Upon execution, the program will output the error message and stop.

```
include
  "caesar_standard.h"
end_include

flag
  "-I/common/Cadp/incl -L/common/Cadp/bin.sun5 -lcaesar"
end_flag

func error (string) : action
  ! implementedby "CAESAR_ERROR"
end_func

error ("Here occurs a fatal error. Farewell.")
```

PROGRAM

The syntax of an XTL program is given by the following grammar:

```
PG  ::=  [ ( FD1 | ETD1 | EFD1 | EID1 | ELD1 )
           ...
           ( FDm | ETDm | EFDm | EIDm | ELDm ) ]
         [ MD1 ... MDp ]
         [ LI1 ... LIr ]
             E
         [ "where"
           [ MDp+1 ... MDp+q ]
           [ LIr+1 ... LIr+s ]
           ( FDm+1 | ETDm+1 | EFDm+1 | EIDm+1 | ELDm+1 )
           ...
           ( FDm+n | ETDm+n | EFDm+n | EIDm+n | ELDm+n ) ]
```

The expression *E* is the body of the program. There may also be lists of function definitions, external directives, macro definitions, and/or library inclusions, placed in the front of the program or at its end, after the "**where**" keyword. The functions defined by *FD1*, ..., *FDm+n* are visible in the body *E* of the program, as well as in all their bodies.

HOW TO CREATE A XTL FILE

At present, XTL files must be written by hand.

HOW TO READ A XTL FILE

At present, there is one single CADP tool, **xtl**(LOCAL), that reads and processes XTL files.

BIBLIOGRAPHY

[MG98] R. Mateescu and H. Garavel. XTL: Meta-Language and Tool for Temporal Logic Model-Checking. Proc. of the International Workshop on Software Tools for Technology Transfer STTT'98 (Aalborg, Denmark). BRICS Notes Series NS-98-4, pp. 33-42, 1998. Available from http://cadp.inria.fr/publications/Mateescu-Garavel-98.html

[GLMS13] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. Springer International Journal on Software Tools for Technology Transfer (STTT), 15(2):89-107, 2013. http://cadp.inria.fr/publications/Garavel-Lang-Mateescu-Serwe-13.html

SEE ALSO

bcg(LOCAL), **bcg_io**(LOCAL), **xtl**(LOCAL).

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

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

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

BUGS

Please report bugs to Radu.Mateescu@inria.fr