

PART 3
CPN ML Reference

Chapter 30

Introduction to CPN ML

Design/CPN uses an extension of Standard ML called *CPN ML* for:

- CP net inscriptions - initial markings, arc expressions and guards.
- Internal algorithms that calculate the enabling and occurrence of bindings.
- Transition code segments.

Standard ML Features

Standard ML includes the following features that support the execution of CP nets:

- **Functional language** - provides a natural and general way to specify the arc expressions and guards.

Moreover, Standard ML makes it very easy to translate a CP graph into a CP matrix because it supports lambda notation.

Finally, it simplifies the calculation of place invariants for CP nets by applying a set of reduction rules formulated in terms of standard operations on functions.

- **Interactive language** - enables users to change individual net inscriptions without having to reevaluate all of them.
- **Strong typing** - allows Design/CPN to detect colorset errors during syntax checking.
- **Polymorphic type system** - permits operations and functions to be used by different colorsets instead of having to declare a version for each colorset.
- **Overloading** - permits operations and functions to be used for several different purposes. For example, the symbol "+"

is used for addition in an integer or real colorset as well as for the addition of multisets and function.

The polymorphic type system and operator/function overloading facilitates the use of standard mathematical notation for net inscriptions.

- **Static scoping** - provides a consistent and stable runtime environment. Since the environment is fixed at runtime, it is easier to understand the functionality of large code segments.

In addition, internal routines are more efficient because identifiers can be bound when the functions are declared instead of when they are evaluated.

- **Exception mechanism** - handles runtime errors. When an error occurs, an exception is raised and the ML system looks for an exception handler. If no system or user-defined exception handler is found, the exception is reported as an error.

CPN ML Extensions to Standard ML

CPN ML extends Standard ML by the addition of three capabilities:

- CPN datatypes, called *colorsets*.
- CPN variables.
- Reference variables with a specified scope.

Colorsets

Colorsets are extensible. This allows CP nets to more accurately reflect the system structures being modeled. Colorsets are described in Chapter 32.

CPN Variables

CPN variables provide CP net components, such as arc inscriptions and guards, with the ability to reference different values. They are characterized by:

- Scope is local to transitions.

- Multiple possible simultaneous bindings on different transitions.
- Extent is the firing of a particular transition.

CPN Variables are described in Chapter 34.

Reference Variables

A page may have many instances. These instances have identical code segments and thus they usually work on the same set of variables.

However, sometimes it is necessary (or at least convenient) to declare local variables where each page instance has its own private version of the variable called a *reference variable*.

CPN ML allows you to declare global, page, and instance reference variables. Chapter 35, “Reference Variables,” describes both CPN reference variables and `val`-defined reference variables.

Declaration Nodes

All colorset, CPN variable, and value declarations (with the exception of value declarations within `let` structures) must be declared in a declaration node.

This is a departure from Standard ML requirements. It means that a value identifier cannot be redefined during the execution of the CP net.

Each syntactically correct CPN diagram can contain exactly one global declaration node and at most one temporary declaration node. In addition, each page can contain at most one local declaration node. Each of the three types of nodes contains *declarations* of items that can be used as *inscriptions* on the diagram.

CPN ML Restrictions and Modifications

Standard ML provides many more capabilities than are actually needed to support CPN modeling, and a few conventions that are not appropriate to the needs of Design/CPN. CPN ML does not implement the unnecessary capabilities, and changes the inappropriate conventions, as described in this section.

CPN ML Reference

Abstract Datatypes

Colorsets cannot be defined in terms of abstract datatypes.

Wildcards

Wildcards are not supported.

Modules

Modules are not relevant to CP Nets and are not supported.

Side Effects

Side effects are only allowed in code segments.

@ Operator (List Concatenation Operator)

In timed simulation CPN ML redefines the @ operator to append time stamps. The list concatenation operator becomes ^^ (double hat).

Conventions

Syntax descriptions

Format

- 1) Format line - The actual form to use when writing a specific operator, function, or constant.

The font is courier.

Words and symbols that must be entered literally are shown in **bold**.

Optional components are shown with angle brackets. As with the required portion, literal words to be entered are shown in **bold**.

In all cases the form terminates in a semicolon, even though sometimes the Design/CPN context does not require it. For example, initial markings do not terminate in a semicolon.

- 2) Description line - Shows the colorset of the constant, or the domain and range colorset for the function or operator.

If any colorset may be used, then *type variables* are used to indicate this fact. Each type variable, usually a lowercase a, is preceded by either a single quote or a double quote.

The *single quote* indicates that any colorset may be used.

The *double quote* indicates that any colorset for which the equality operation is defined may be used. In practice this means that any colorset may be used since currently all colorsets have an equality operation defined for them.

If either the domain or the range is the Cartesian product of two colorsets, then it is shown as:

$$(\text{colorset}_1 \bullet \text{colorset}_2)$$

Examples

Examples provide one or all of the following:

- 1) Declarations, if any, required to understand the example
- 2) Expression(s) actually entered, preceded by an ML> prompt.
- 3) The value returned by the system is in the form returned by Design/CPN after evaluation, preceded by a single >.

Testing Examples

To test non-CPN inscription examples:

- 1) If there are any declarations, create a global or temporary declarations box with the declarations required and then syntax check it with the **Syntax Check** command.
- 2) Create an aux box with the **box** command (**Aux** menu).
- 3) Type in the expressions exactly as shown in the example.
- 4) Select the expressions and evaluate it with the **ML Evaluate** command (**Aux** menu).
- 5) The result is shown in a separate evaluate box and should match the results shown in the example.

To test CPN inscription examples:

See the User Guide for CP net creation techniques.

Chapter 31

Identifiers

CPN ML identifiers are alphanumeric sequences of letters, digits, primes (apostrophe), and underscores starting with a letter.

They are used as the names for:

- Colorsets (Chapter 32)
- Record colorset field labels (Chapter 32)
- Value constructors (Chapter 33)
- CPN variables (Chapter 34)
- Reference variables (Chapter 35)
- Operators and function symbols (Chapters 36 & 38)

Examples

These identifiers are legal:

SmallOrder

e3'f4

big_bertha

These are illegal, because they do not begin with a letter:

3e4f

_New_Id

Reserved Words

Reserved words are words that have a predefined meaning for CPN ML. They may not be used as identifiers:

```
abstype action all and andalso as bool
by case color colorset datatype declare do
else end exception fn fun globref guard
handle if in index infix infixr input
instref int let list local ms nonfix of
op open orelse output pageref product
raise real rec record ref same start
string subset then time TIME tms type unit
untimed val value var while with ( ) [
] { } , : := ; .. ... ` | | ? @ ! = => -> _
#
```

Identifier Duplication

The syntax check in Design/CPN will usually not allow CPN ML identifiers to be identical to each other. However, there are exceptions:

- **Record labels** - allowed to be identical to each other if they appear in different colorset declarations. Record labels are also allowed to be identical to other kinds of CPN ML identifiers, with the exception of CPN variables. See Chapter 32 for a discussion of colorsets.
- **Page/instance reference variables** - allowed to have identical identifiers if their scopes do not overlap (they are declared on different pages).

Page/instance reference variables are also allowed to have the same name as global reference variables: the page/instance reference variable overwrites the global reference variable at the page where the page/instance reference variable is declared. See Chapter 35 for a discussion of reference variables.

Predeclared Identifiers

In addition to the reserved words which may not be used as identifiers, Design/CPN uses a number of predeclared identifiers:

- Predeclared constants, operations and functions.

- Predeclared exceptions (all of these begin with “CPN' ” or end with an apostrophe (’), also called a prime, and the name of the colorset).
- Constants, variables and functions in the internal algorithms (all of these begin with “CPN' ”).
- Internal representation of multisets and bindings (applies the identifiers “!!”, “!!!”, “\$”, “\$\$” and “\$\$\$”).

All the predeclared identifiers are described in Chapter 41. You can safely reuse these identifiers unless they are marked for internal use by a star in tables and descriptions in that chapter.

Comments

Comments are identified by an initial (* and a terminating *). Comments may be inserted anywhere that white space can appear in CPN net inscriptions. For example,

```
(* Enumerated colorset *)

color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

(* Indexed colorset *)

color BankBranch =
    index branch_number with 1...8;

color integer = (* Integer colorset *) int;
```

Since comments are treated as white space, inserting them in the middle of identifiers causes a syntax error. For example,

```
var New_int : inte(* comment *)ger;
```

results in the following error message:

```
Error:
Syntax error in variable declaration:
inte ger;
```


Chapter 32

Colorsets

Design/CPN provides a mechanism for user-defined datatypes. For historic reasons these user-defined datatypes are called *colorsets*.

Each colorset represents a set of values and is identified by an alphanumeric identifier.

The colorsets must be declared in global, local, and temporary declaration nodes. Design/CPN automatically generates a set of pre-declared constants, operations, and functions for each colorset declared.

Classifying Colorsets

Colorsets are classified by size and by complexity.

Size

Colorsets can be classified as large or small. This distinction determines which optional system-defined constants, operations, functions may be declared for a particular colorset.

A colorset is large if it contains too many elements to enumerate. Otherwise, it is small:

- Unit, bool, indexed and enumerated colorsets are small.
- Int, and string colorsets are small if and only if they include the appropriate **with** specification. Note that the real colorset is always large, even with the **with** specification.
- Product, record, and union colorsets are small if and only if all their component colorsets are small.
- List colorsets are small if and only if their component colorset is small and they are declared by means of a **with** clause.

- Subset colorsets are small if and only if either their component colorset is small or their subset specification is a list.

Complexity

A colorset is *compound* if it is constructed from other colorsets. Otherwise, it is *simple*.

Simple Colorsets

Unit

The unit colorset comprises a single element, denoted (). It has no predefined operations that use it, but is very useful as a placeholder. See examples in Chapter 40, “Inscriptions.”

Declaration Syntax

```
color colorset_name = unit <<with
    new_unit_id>>;
```

Optional With Clause

Renames the value, that is, defines the identifier representing (). The new value name must be an alphanumeric identifier.

Declaration Example

```
color terminator = unit with nothing_here;
```

Value Representation Example

```
ML> nothing_here;
> () : terminator
```

Boolean

The boolean values are the constants **true** and **false**.

Declaration Syntax

```
color colorset_name = bool <<with
    (new_false, new_true)>>;
```

Optional With Clause

Renames the values, that is, defines the identifiers representing “true” and “false”. The new value names must be alphanumeric identifiers.

Declaration Example

```
color answer = bool with
    (no, yes);
```

Value Representation Example

```
ML> no;
    > false : answer
```

Boolean Operations

The following operations are predefined for booleans:

not The unary negation operation.

For example:

```
ml> not true;
    false : bool
```

This example returns false, which is the negation of the boolean value true.

andalso Boolean conjunction. And.

orelse Boolean disjunction. Inclusive or.

relational

> Greater than

< Less than

>= Greater than or equal

<= Less than or equal

<> Not equal

= The equality operator. All the colorsets, with the exception of functions have their own infix equality operators that return boolean values.

if, then, else

The conditional expression. See Chapter 38, “Functions,” for a discussion of its use.

Boolean Selectors

See Chapter 38, “Functions,” for a discussion of their meanings and use.

Integers

Integers are numerals without a decimal point. The integer colorset is large unless restricted by the **with** clause, in which case it is small.

Declaration Syntax

```
color colorset_name = int <<with
    int-expstart .. int-expend>>;
```

Optional With Clause

Restricts the integer colorset to an interval determined by the two expressions `int-expstart` and `int-expend`.

The expression `int-expstart` must be `int-expend`.

Declaration and Use Example

```
color small_integer = int with 1..3;

var SmallInt : small_integer;
```

The CPN variable `SmallInt` may have a range of integer values from 1 to 3. For example, 2 is a legal value, but 4 is not and 2.9 is not. See Chapter 34 for a discussion of the creation and use of CPN Variables.

Integer Operations

The following operations are predefined for integers:

abs	The unary absolute value operator.
~	The unary negation operator.
+	The infix addition operator.
-	The infix subtraction operator.
*	The infix multiplication operator.

Division is handled by two operations:

div	Quotient
mod	Remainder

For example:

```
ML> 37 div 5;
      7 : int
ML> 37 mod 5;
      2 : int
```

Real Numbers

Real numbers are distinguished from integers by the decimal point. The period separates the integer part from the fractional part. One or more digits must follow after the decimal point

The number can be written in scientific notation where the power of 10 is given after the exponent symbol E. The fraction part is optional if the exponent part is present. The real colorset is large.

Declaration Syntax

```
color colorset_name = real <<with
                        real-exp_start..real-exp_end>>;
```

Optional With Clause

Restricts the real colorset to an interval determined by the two expressions `real-expstart` and `real-expend`.

The expression `real-expstart` must be `real-expend`.

Declaration and Use Example

```
color small_real = real with 1.0..3.0;
var SmallReal : small_real;
```

The CPN variable `SmallReal` may have a range of real values from 1.0 to 3.0. For example, 1.0 and 2.9 are legal values but 1 and 4.9 are not.

Real Operations

The following operations are predefined for reals:

<code>abs</code>	Absolute value (infix)
<code>~</code>	Negation (prefix)
<code>+</code>	Addition (infix)
<code>-</code>	Subtraction (infix)
<code>*</code>	Multiplication (infix)
<code>/</code>	Division (infix)
<code>sqrt</code>	Square root (unary)
<code>ln</code>	Natural logarithm (unary)
<code>exp</code>	Exponential (unary)
<code>sin</code>	Sine (unary)
<code>cos</code>	Cosine (unary)
<code>tan</code>	Tangent (unary)
<code>arctan</code>	Arc tangent (prefix)

The trigonometric functions all work in units of radians.

Strings

Strings are specified by sequences of printable ASCII characters surrounded with double quotes. The string `colorset` is large unless restricted by the **with** clause, in which case it is small.

Characters are strings of length one.

Declaration Syntax

```
color colorset_name = string <<with
    string-expb .. string-expt and
    int-expb .. int-expt >>;
```

Optional With Clause

Restricts the character set of string colorsets.

The character set is specified by the string expressions

```
string-expstart .. string-expend.
```

Each string expression must evaluate to a single character (characters are defined as strings of length one) and

```
string-expstart must be string-expend.
```

Optional And Clause

Restricts the length of string colorsets.

The minimum and maximum length of the string is specified by the

```
integer expressions int-expstart .. int-expend.
```

The integer expressions must be:

```
0 int-expb int-expt.
```

Declaration and Use Example

```
color small_string = string
    with "a".. "d" and 3..9;
var SmallString : small_string;
```

The CPN variable `SmallString` may contain only the letters a, b, c, and d. Its length must be 3 and 9. For example, "abcd" and "bdbdbbdb" are legal values but "bcde" and "ab" are not.

```
ML> "abc"
> "abc" : string
```

String Operations

size	length of a string (prefix)
^	concatenate strings (infix)
ord	octal code word corresponding to a character (prefix)
chr	character corresponding to an octal code word (prefix)
mkst_ms	string representation for a multiset
mkst_col	string representation for a colorset

String Compare

Lexigraphic comparison based on the ASCII code representation of the string. The comparison is based on the first character pair that differs in the string.

>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
=	equal
<>	not equal

Escape Sequences

The backslash operator provides a way to insert control characters in strings.

\n	line feed
\t	tab

Enumerated Values

Enumerated values are explicitly named as identifiers in the declaration. These values must be alphanumeric identifiers. This is a small colorset.

Declaration Syntax

```
color colorset_name =
    with id0 | id1 | ..... | idn;
```

Declaration Example

To represent several banks operating in Massachusetts:

```
color Banks = with HarvardTrust |
    Shawmut | NationalGrand;
```

Value Representation Example

A particular bank is then simply entered:

```
ML> HarvardTrust ;
    > HarvardTrust : Banks
```

Suggested Uses

Typically used for a limited set of predefined values.

Indexed Values

Indexed values are sequences of values comprised of an identifier and an index-specifier. The index-specifier range is:

```
int-expstart .. int-expend
```

When:

```
int-expstart and int-expend are integers,
```

and:

```
int-expend is 0
```

and:

```
int-expstart is int-expend
```

the colorset is small.

Declaration Syntax

```
color colorset_name = index identifier
                    with int-expstart .. int-expend;
```

Declaration Example

To represent the different branches in a bank:

```
color HarvardTrust_bank = index branch_number
                        with 1..8;
```

Value Representation Example

```
ML> branch_number(3);
> branch_number 3 : HarvardTrust_bank
```

Suggested Uses

Typically used for naturally indexed values.

Compound CPN Colorsets

The following colorsets use previously declared colorsets.

Tuples

A fixed-length, positional colorset whose set of values is identical to the cartesian product of the values in previously declared colorsets. Each of the component colorsets may be a different type.

Declaration Syntax

```
color colorset_name= product
                    colorset_name1*colorset_name2*
                    ...*colorset_namen;
```

n must be ≥ 2 for colorset_name₁, colorset_name₂, ..., colorset_name_n.

The values in the set are tuples. Individual values are positionally accessed.

Declaration Example

A business rule that large customer orders must be processed by expert staff members, where the customer order colorset is declared by:

```
color customer_order_size = with Big | Medium
    | Small;
```

and the staff expertise is declared by:

```
color staff_type = with Expert | Journeyman |
    Novice
```

can be represented by a tuple colorset:

```
color order_process_requirement =
    product customer_order_size * staff_type;
```

Representation Examples

```
ML> (Big,Expert);

> (Big,Expert) : customer_order_size
    * staff_type
```

Note: This example uses explicit values. If the tuple had contained CPN variables it could have been used as a pattern. See Chapter 36, “Expressions,” for a discussion of tuple patterns.

Suggested Uses

Used in the collecting of data where the values are tuples.

Records

A fixed-length colorset whose set of values is identical to the cartesian product of the values in previously declared colorsets. Each of the component colorsets may be a different type and each is identified by a unique label so that each field is position-independent.

Declaration Syntax

```
color colorset_name= record
    id1:colorset_name1 * id2:colorset_name2 *
    ...* idn:colorset_namen;
```

CPN ML Reference

n must be 2 for colorset_name₁, colorset_name₂, ..., colorset_name_n.

The values in the set are labeled records and their contents can be recognized by the fieldname.

Selector Functions

For records, CPN ML contains a set of predeclared selector functions that access components of the tuple:

```
#id1, #id2, ..., #idn.
```

Declaration Example

A business rule that large customer orders must be processed by expert staff members, where the customer order colorset is declared by:

```
color customer_order_size = with Big | Medium  
| Small;
```

and the staff expertise is declared by:

```
color staff_type = with Expert | Journeyman |  
Novice
```

can be represented by a record colorset:

```
color order_process_requirement =  
record Order:customer_order_size  
* Staff:staff_type;
```

Value Representation Example

```
ML> {Order=Big, Staff=Expert};  
  
> {Order=Big, Staff=Expert} :  
{ Order:customer_order_size,  
Staff:staff_type }
```

Note: This example uses explicit values. If the record had contained CPN variables it could have been used as a pattern. See Chapter 36, "Expressions," for a discussion of record patterns.

Selector Example

```
ML> #Order{Order=Big, Staff=Expert};
    > Big : customer_order_size
```

Suggested Uses

Used in the collecting of data where the values are records.

Lists

A variable-length colorset. The values are a sequence whose colorset must be the same type.

Declaration Syntax

```
color colorset_name = list colorset_name0
    <<with int-expb .. int-expt>>;
```

Optional With Clause

The **with** clause specifies the minimum and maximum length of the lists.

Declaration Example

The asset types for a bank could be maintained in a list where the assets have been declared as strings:

```
color asset_list = list assets;
```

Value Representation Example

```
ML> ["real estate", "treasury bonds"] ;
    > ["real estate", "treasury bonds"] :
    string list
```

Note: This example uses explicit values. If the list had contained CPN variables it could have been used as a pattern. See Chapter 36, “Expressions,” for a discussion of list patterns.

List Operators and Functions

Prepend: The `::` operator creates a list from an element and a list by placing the element at the head of the list:

```
ML> 1::[2,3,4] ;
      > [1,2,3,4] : int list
```

If the list is the empty list, the result is a list of one member. For example:

```
ML> 1::[] ;
      > [1] : int list
```

This is one way to get list construction started.

Concatenate: The `^^` operator creates a list from two source lists. The elements of each list are extracted and combined to form the new list:

```
ML> [1,2,3]^^[4,5,6] ;
      > [1,2,3,4,5,6] : int list
```

To create a new list whose elements are the source lists, not the elements of the source list, put an extra set of brackets around each list:

```
ML> [[1,2,3]]^^[4,5,6];
      > [[1,2,3],[4,5,6]] : (int list) list
```

This is the same as if you had said:

```
ML> [1,2,3]::[[4,5,6]];
      > [[1,2,3],[4,5,6]] : (int list) list
```

because the element colorset in this example is an integer list.

Suggested Uses

Using one token with a list of values gives access to all the values at once and makes ordering and other behavior "based on all tokens" possible.

Subsets

Subsets are a selection of values in a previously declared colorset.

Declaration Syntax

The subset specification can take two different forms, function and list:

Function Form

```
color colorset_name = subset colorset_name0
    by subset-function;
```

By Clause

Specifies a function whose return value is a boolean. `colorset_name` will contain exactly those values from `colorset_name0` that are mapped into the boolean value true.

List Form

```
color colorset_name = subset colorset_name0
    with subset-list;
```

With Clause

Specifies a list with elements from `colorset_name0`. `colorset_name` will contain exactly those values that are listed.

Declaration and Use Example - List Form

```
color Banks = with HarvardTrust |
    Shawmut | NationalGrand;

color Banksub = subset Banks
    with [Shawmut, NationalGrand];

ML> [Shawmut, NationalGrand];

> [Shawmut, NationalGrand] : Banks list
```

Notice that when simply given as a value list CPN ML responds with a combination colorset - Banks list. CPN ML would have similarly accepted the following:

```
ML> [HarvardTrust, NationalGrand];  
    > [HarvardTrust, NationalGrand] :  
      Banks list
```

However, if `Banks` is the colorset for a place, Design/CPN would enforce the subset and would reject an inscription of `[HarvardTrust, NationalGrand]` as a syntax error.

Suggested Uses

Restriction of an already defined colorset. Implies runtime checks, initial, and current markings.

Union

union is a (disjoint) union of previously declared colorsets.

Declaration Syntax

```
color colorset_name =  
  union id1<<:colorset_name1>> +  
        id2<<:colorset_name2>>  
  + ..... + idn<<:colorset_namen>>;
```

For each component colorset, provide a unique selector. This construction is similar to variant records in other programming languages.

If `colorset_namei` is omitted, then `idi` is treated as a new value, and it can be referred to simply as `idi`.

The sequence of selectors in the declaration defines the order of elements in the new colorset. For the predeclared function `lt'colorset_name`, this implies that any value with selector `idi` is less than any value with selector `idj`, provided that $i < j$.

For a given selector, `idi`, the ordering of values is defined by `lt'colorset_namei`.

The predeclared function `ran'colorset_name` works as follows:

- first a selector `idi` is drawn randomly,
- then the function `ran'colorset_namei` for the corresponding component colorset (if any) is applied.

Declaration Example

```

color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

color No_of_Branches = int with 0..10;

color No_of_Customers = int with 0..99999;

color RegionBanksSize = union Name:Banks +
                             CustomerBase : No_of_Customers +
                             BranchNumbers : No_of_Branches;

```

See Chapter 40, “Inscriptions,” for an example showing arc inscriptions based on this colorset.

Suggested Uses

A union colorset is similar to a variant record in other programming languages with selectors to signify variant type.

Alias CPN Colorset

A colorset that provides an alias construct. The colorset has exactly the same values and properties as a previously declared colorset. However, each of the two colorsets has its own, possibly different, set of predeclared constants, operations, and functions as specified by the optional **declare** clause. Thus, to obtain the same declare set as the source colorset, specify `declare same`.

Declaration Syntax

```

color colorset_name = colorset_name0
                    declare same;

```

The type checking mechanism in Design/CPN does not distinguish between the original colorset and its duplicates. Hence colorsets for all members in a place fusion set must have the same prime colorset. The same rule applies for port nodes and their assigned socket/parameter nodes.

Declaration Example

If two banks are merging and want to describe the same internal structure but distinguish their origin, they could define it:

```

color BayBank_bank = HarvardTrust_bank;

```

Suggested Uses

To prevent inconvenience when colorsets are needed that differ only in name.

Function

See Chapter 38, "Functions."

Declare Clause

The **declare** clause makes the system constants, operations, and functions described below available to the colorset.

The syntax is:

```
color colorset declare id1, ..., idk;
```

id₁, ..., id_k is a comma-separated sequence of the desired names. Two special keywords have been defined to ease the declaration:

- **declare all** - yields all possible constants, operations, and functions for the actual kind of colorset.
- **declare same** - declares for a duplicate colorset the same functions that were requested for the original colorset.

Random Value

Returns a random value. The colorset of the returned value is that specified in the function call.

Syntax

```
ran'colorset ();
```

Function from: unit -> colorset.

The function cannot not be used in net inscriptions such as arc inscriptions and guards. It can, however, be used in time regions and in code segments.

Example

Given the following colorset declaration:

```
color Banks =
  with HarvardTrust | Shawmut |
  NationalGrand;
```

then the following function call:

```
ran'Banks ();
```

returns one of the enumerated values. One response might be:

```
> Shawmut : Banks
```

Another time the response might be:

```
> NationalGrand: Banks
```

Less Than

Tests whether a value in the specified colorset is less than another value in same colorset with respect to the colorset ordering. (See the section on colorset ordering at the end of this chapter).

Syntax

```
lt'colorset (value-identifier1,
            value-identifier2);
```

Function from: (colorset * colorset) -> bool.

Example

Given the following colorset declaration:

```
color Banks =
  with HarvardTrust | Shawmut |
  NationalGrand;
```

then the following function call:

```
lt'Banks (Shawmut, NationalGrand);
```

returns:

```
> true : bool
```

String Representation of a Value

Returns the string representation for a value of a particular colorset.

Syntax

```
mkst_col'colorset (value-identifier);
```

Function from: colorset -> string.

To get around the limited input line length in the Edinburgh interpreter (256 characters), a line feed is inserted after each element in a list and in a record by `mkst_col` for list and record colorsets, respectively.

Example

Given the following colorset declaration:

```
color Banks =  
  with HarvardTrust | Shawmut |  
  NationalGrand;
```

then the following function call:

```
mkst_col'Banks (Shawmut);
```

returns:

```
> "Shawmut" : string
```

String Representation of a Multiset

Returns the normalized string representation for a multiset of a particular colorset.

Syntax

```
mkst_ms'colorset (multiset-specifier);
```

Function from: colorset ms -> string.

Example

Given the following colorset declaration:

```
color Banks =  
  with HarvardTrust | Shawmut |  
  NationalGrand;
```

then the following function call:

```
mkst_ms'Banks (1`Shawmut + 2`HarvardTrust);
```

returns:

```
> "2`HarvardTrust + 1`Shawmut" : string
```

Multiset, Size, First, and Last Constants

These constants can only be declared for small colorsets. They define:

- 1) The full multiset over the `colorset`. The multiset contains exactly one occurrence of each colorset value.
- 2) The number of values in `colorset`.
- 3) The first value in `colorset`.
- 4) The first value in `colorset`.

ms

Multiset constant declaration

```
<<colorset declaration >> declare ms;
```

Multiset constant use

To create a multiset for the colorset with **ms** declared, either:

Specify the colorset name to obtain the internal representation of the multiset, or

Use the function **mkst_ms** to obtain a string representation of the multiset.

size

```
size 'colorset ;
```

Function from: colorset ->integer-value

first

```
first 'colorset ;
```

Function from: colorset ->identifier-value

last

```
last 'colorset;
```

Function from: colorset ->identifier-value

Examples

Given the following colorset declaration:

```
color Banks = with HarvardTrust | Chase
               | MarbleheadSavings | Shawmut
               | NationalGrand
declare ms, size, first, last;
```

then evaluation of the following :

```
Banks;
mkst_ms'Banks (Banks);
size'Banks;
first'Banks;
last'Banks;
```

displays:

```
> !! ((1,HarvardTrust),!! ((1,Chase),!!
      ((1,MarbleheadSavings),!! ((1,Shawmut),!!
      ((1,NationalGrand),empty)))) : Banks ms

> "1`HarvardTrust + 1`Chase +
   1`MarbleheadSavings +
   1`Shawmut + 1`NationalGrand" : string

> 3 : int

> HarvardTrust : Banks

> NationalGrand : Banks
```

Ordinal Number and Value

ord and **col** can only be declared for enumerated and indexed colorsets. They

- 1) Convert from a value to a number representing its position in the colorset definition.
- 2) Convert from a number representing a value's position in the colorset definition to the value.

The result of **ord**'colorset is always in the interval:
0..(size'colorset-1).

The argument of `col 'colorset` should be in the same interval; otherwise, an exception is raised.

Syntax

```
ord'colorset (identifier_value);
```

Function from: colorset -> int

```
col'colorset (position_value);
```

Function from: int -> colorset

Examples

Given the following colorset declaration:

```
color Banks =  
  with HarvardTrust | Shawmut | NationalGrand  
  declare ord, col;
```

then evaluation of the following :

```
ord'Banks (HarvardTrust);  
col'Banks (2);
```

displays:

```
> 1 : int  
> Shawmut : Banks
```

Index Number and Value

index and **clr** convert from index number to identifier-value, and vice versa. They can only be declared for indexed colorsets.

The argument to **index** is the identifier-value specified in the colorset definition followed by an index number in parentheses. The value returned is the integer index number.

The range of the argument to **clr** must be that of the index numbers specified in the colorset definition; otherwise, an exception is raised.

Syntax

```
index'colorset (identifier_value  
  (index_number));
```

Function from: colorset -> int

```
clr'colorset (index_number);
```

Function from: int -> colorset

Examples

Given the following colorset declaration:

```
color FleetBank_branch = index branches with  
1..8 declare index, clr;
```

then evaluation of the following :

```
index'FleetBank_branch (branches(1));  
clr'FleetBank_branch (3);
```

displays:

```
> 1 : int  
> branches 3 : FleetBank_branch
```

Distance and Rotation for Values

dist and **rot** provide cyclic manipulation of the values of colorset. They can only be declared for enumerated and indexed colorsets.

dist returns the distance from the first value to the second in the forward direction. The result is always in the interval:
0..(size'colorset-1).

rot returns a value, which is reached from a given value by rotating a specified distance, positive or negative. There is no restriction upon the integer argument of rot'colorset.

Syntax

```
dist'colorset (identifier_value1,  
               identifier_value1);
```

Function from: (colorset * colorset) -> int

```
rot'colorset distance-to-rotate  
identifier_value;
```

Function from: int -> (colorset -> colorset)

Examples

dist example

Given the following colorset declaration:

```
color Banks =
  with HarvardTrust | Shawmut | NationalGrand
  declare dist, rot;
```

then evaluation of the following :

```
dist'Banks (HarvardTrust, NationalGrand);
```

displays:

```
> 2 : int
```

rot example

Given the following colorset declaration:

```
color Banks =
  with HarvardTrust | Shawmut | NationalGrand
  declare dist, rot;
```

then evaluation of the following :

```
rot'Banks 2 Shawmut;
```

displays:

```
> HarvardTrust : Banks
```

Multiplication of Multisets

mult constructs a multiset for a tuple or record colorset by multiplying a set of multisets constructed from its component colorsets.

Syntax

```
mult'colorset (colorset1 ms * colorset2 ms *
..... * colorsetn ms);
```

Function from: (multiset*multiset...)-> multiset

Example

Given the following colorset declarations:

```
color alpha = with a | b | c;  
color number = with num1 | num2 | num3;  
color tuple = product alpha * number declare  
  mult;
```

then evaluation of the following :

```
mult'tuple ((1`a) + (3`c), (3`num1) +  
  (4`num3));
```

displays the internal multiset representation:

```
> !! ((3,(a,num1)),!! ((4,(a,num3)),  
  !! ((9,(c,num1)),  
  !! ((12,(c,num3)),empty))) : tuple ms
```

To display the multiset as a string:

```
mkst_ms'tuple (mult'tuple ((1`a) + (3`c),  
  (3`num1) + (4`num3)));
```

displays:

```
> "3`(a,num1) + 4`(a,num3) + 9`(c,num1) +  
  12`(c,num3)" : string
```

Membership of a Subset CPN Colorset

in tests whether a given value is a member of the colorset.

This function can only be declared for subset colorsets or for `int`, `real` or `string` colorsets defined using a **with**-clause specification.

Syntax

```
in'colorset (value);
```

Function from: colorset -> bool

Example

```
color Banks = with HarvardTrust | Chase  
  | MarbleheadSavings | Shawmut  
  | NationalGrand;
```

```

color LocalBanks = subset Banks
                    with [NationalGrand,
                        MarbleheadSavings];

ML> in'LocalBanks (NationalGrand);

> true : bool

```

Membership of Union Base CPN Colorset

of_id_i tests whether a value from a union colorset belongs to a specific component colorset (with the selector id_i).

Syntax

```
of_idi'colorset (component_colorset);
```

Function from: colorset -> bool

Example

```

color Banks = with HarvardTrust | Chase
                | MarbleheadSavings | Shawmut
                | NationalGrand;

color No_of_Branches = int with 0..10;

color No_of_Customers = int with 0..99999;

color RegionBanksSize = union Name:Banks +
                            CustomerBase : No_of_Customers +
                            BranchNumbers : No_of_Branches
declare of_CustomerBase,
        of_BranchNumbers;

ML> of_CustomerBase'RegionBanksSize
      (CustomerBase 8);

> true : bool

ML> of_CustomerBase'RegionBanksSize
      (CustomerBase 50000000);

> false : bool

ML> of_BranchNumbers'RegionBanksSize
      (BranchNumbers 50000000);

> false : bool

```

Ordering of CPN Colorsets

All colorsets are ordered. The order is defined in the following way:

enumerated	Determined by the order in which the value names appear in the colorset declaration.
indexed	Determined by the usual integer ordering of the indexes.
product	Lexicographic (with the ordering of the base colorsets).
record	Lexicographic (with the ordering of the base colorsets).
list	Lexicographic (with the ordering of the base colorset).
subset	Determined by the order of the base colorset.
unit	Trivial order.
bool	False before true.
int, real	Usual ordering of numbers.
string	Lexicographic (with the ASCII ordering).

For non-predefined ML colorsets, the order is determined from the `lt` function in the additional specification.

Chapter 33

Values

Overview

A value declaration connects an identifier to a value. The reserved word **val** is used to declare an identifier and an arbitrary expression may be used as the value.

Values are not just constants - functions are declared as values (the reserved word **fun**, used to declare functions, returns a value declaration). Function values are discussed in Chapter 38, "Functions."

In CPN ML values must be declared in a declaration node. They may be used but not redefined in inscriptions and in code segments. The exception is values that are declared in a `let` structure.

They are frequently used to define constants used in multiple locations as well as for specifying time delays.

Declaration

Syntax

The syntax is:

```
val identifier = expression;
```

where `identifier` is an identifier and `expression` is a CPN ML expression, including multiset expressions. The expression represents the value to be associated with the identifier.

Value Representation

Val declaration expressions may use any value representation that is syntactically unique, without a prior colorset declaration. These include:

- unit
- bool
- int
- real
- string
- tuple
- list
- record

Examples:

These examples all use the conventions described in Chapter 30, "Introduction to CPN ML," in order to show the result of the declaration. In Design/CPN practice, however, they would be declared in a declaration node and no value would be displayed.

Integer value

```
ML> val intval = 3;  
> val intval = 3 : int
```

Boolean value

```
ML> val booleanval = true;  
> val booleanval = true : bool
```

Real value

```
ML> val realval = 3.0;  
> val realval = 3.0 : real
```

String value

```
ML> val stringval = "string 3.0";  
> val stringval = "string 3.0" : string
```

Tuple value

```
ML> val tupleval = (3,4.0);  
> val tupleval = (3,4.0) : int * real
```

List value

```
ML> val listval = [3,4];
> val listval = [3,4] : int list
```

Colorsets

Value declarations may use expressions of previously declared colorsets. For example, the following declaration is acceptable because the colorset of the value of `bigjob` has been previously declared:

```
color Staff =
  with Expert | Journeyman | Novice;

color ProcessOrder = product Order * Staff;

val bigjob = (Big, Expert);
```

The following example is also acceptable:

```
val integer_tuple = (1, 4);
```

because the `(1, 4)` specification is one of the syntactically unique value representations known to Design/CPN.

The following example, however:

```
val alpha_tuple = (ss, aa);
```

produces the following error message:

```
Errors

C.4 Error in global/temporary declaration node

Type checking error in: ss
Unbound value identifier: ss
[Closing <string>]
[Closing <string>]
« «18» »
```

because the system doesn't recognize the value `(ss, aa)`. If `ss` and `aa` were specified as strings:

```
val alpha_tuple = ("aa", "ss");
```

the declaration would be acceptable.

Other types of specifiers

Multiset Expression

Multiset expressions may be used as value specifiers. For example:

```
ML> val baz = 1`3 + 3`4;  
  
> val baz = !! ((3,4),!! ((1,3),empty)):int  
ms
```

See Chapter 37, “Multisets,” for a complete description.

Function expression

Function expressions may be used as value specifiers. For example:

```
ML> val baz = fn x => 3 + x;  
  
> val baz = fn : int -> int
```

See Chapter 38, “Functions,” for a complete description.

Value Uses in CPN Inscriptions

Chapter 40, “Inscriptions,” describes the use of constants in CPN inscriptions.

Chapter 34, “CPN Variables,” describes the Design/CPN mechanism for representing changing values in CPN inscriptions.

Chapter 34

CPN Variables

Term Definitions

Variables

A *variable* is an identifier whose value can be changed during the execution of the model.

Binding

Binding is the association of a value with a variable. A binding has both scope and extent.

Scope

Scope is the locations in a model in which a particular binding can be referenced.

Extent

Extent is the interval during which a particular binding is in effect.

CPN Variables

CPN variables are variables that are used in CP net inscriptions. They have the following characteristics:

- They are declared using the reserved word **var** and the name of a previously declared colorset in a global or temporary declaration node.
- They are bound to a variety of different values from their colorset by the simulator as it attempts to determine if a transition is enabled.

- There can be multiple bindings simultaneously active on different transitions. These bindings can exist simultaneously because they have different scopes.
- The extent of a CPN variable binding is the firing of a particular transition.
- They provide arc inscriptions with the ability to reference different values. See Chapter 33, “Values,” for a discussion of constants and their use in arc inscriptions.

Declaration syntax

```
var id1, id2, ....., idn : colorset_name;
```

where:

`id` is an alphanumeric identifier,

`:` is the syntactic separator between the identifier(s)

`colorset_name`

is the name of a previously declared CPN colorset.

The system checks whether any of the CPN variable names have been used for type constructors, for example, in a colorset declaration. If an error occurs, a warning is given.

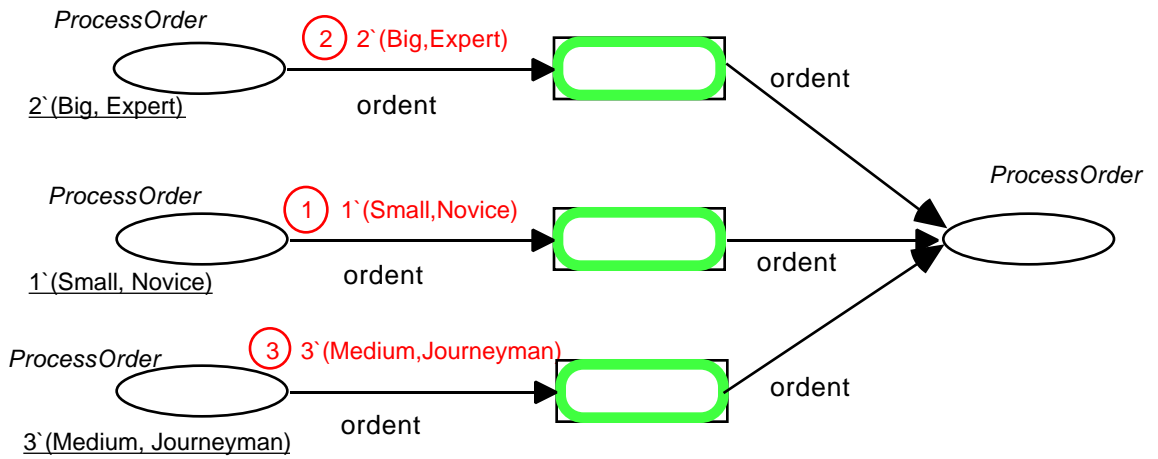
See Chapter 32, “Colorsets,” for a discussion of the declaration and use of CPN colorsets.

Example of Declaration and Use

This example illustrates the significance of the scope characteristics of CPN variables. The following CPN colorset and CPN variables are declared in the global declaration node:

```
color Order = with Big | Medium | Small;  
color Staff = with Expert | Journeyman | Novice;  
color ProcessOrder = product Order * Staff;  
var ordent : ProcessOrder;
```

The following three transitions are all enabled with different values for the CPN variable `ordent`:



Chapter 35

Reference Variables

Reference Variables

ML reference variables are the same thing as pointer variables in imperative languages such as C. They may be used only in code segments.

Reference variables must never be used in any way that affects transition enablement. To illustrate the problems that could arise if this rule were broken, consider the following example:

A reference variable, `refvar`, is used in the arc inscription of two different transitions, transition A and transition B. Both are enabled, and transition A fires. Transition A changes the value of `refvar` in such a way that transition B is no longer actually enabled. But the system has no way of knowing this, since enabled transitions are not rechecked before they are fired (to recheck each time a transition is chosen for firing would degrade performance to the point of unusability). Attempting to fire transition B results in unpredictable system errors.

Reference variables are necessary for accessing information outside the model, such as files and databases, and for providing information about the model. They are used in this manual to handle statistical variables and I/O access.

Val-defined Reference Variables

Val-defined reference variables are defined in declaration nodes. The value stored at the location named by the identifier is accessed a syntax that tells the system to obtain a value pointed to be a name, not to change the value of the name itself.

Caution

Design/CPN is unable to check whether a reference variable has been declared with **val** instead of **globref**. If a reference variable declared by means of **ref** has the same name as another reference

CPN ML Reference

variable declared by means of **globref**, **pageref**, or **instref**, the CPN Simulator will apply the latter.

Declaration Syntax

To declare a val-defined reference variable and provide an initial value:

```
val id = ref data_value;
```

where <id> is an alphanumeric identifier, and the value following **ref** is the initial value. The colorset of the initial value is either predefined (See Chapter 33, "Values," for a discussion of predefined identifiers) or has been previously declared with a **color** declaration.

For example,

```
val foo = ref (3,4);
```

If this form were interactively evaluated by ML, the system would return:

```
> val foo = ref (3,4) : (int * int) ref
```

which declares the identifier `foo` as tuple - (int * int) - reference variable with the value of (3,4).

Reference Syntax

To update the value of `foo` by 1, type:

```
foo := !foo + 1;
```

accesses the value (which is 0) at the location `foo` and updates it by 1. The `:=` and `!` syntax specifiers tell the system to access the value at the location identified by the name `foo`.

Value Syntax

Entering the reference variable name by itself:

```
foo;
```

results in:

```
> ref 2 : int ref
```

which says that the identifier is a reference variable whose value is two and whose colorset is integer reference.

CPN Reference Variables

Global Reference Variables

CPN global reference variables provide the same capabilities as do the val-ref defined reference variables: they define reference variables with a global scope.

Declaration syntax

```
globref id = exp;
```

Global reference variables must be declared in a global or temporary declaration node and can be used in all code segments in the entire diagram.

The colorset of the global reference variable is determined by the colorset of the expression, which also determines the initial value.

Page Reference Variables

Defines reference variables with a page scope.

Declaration syntax

```
pageref id = exp;
```

Page variables must be declared in a local declaration node and can only be used in code segments on the corresponding page. All page instances of the page with the local declaration node refer to the same reference variable.

The colorset of the expression determines the colorset of the reference variable, which in turn determines the initial value.

Instance Reference Variables

Defines reference variables with an instance scope.

Declaration syntax

```
instref id = exp;
```

Instance variables must be declared in a local declaration node and can only be used in code segments on the corresponding page. Each page instance of the page with the local declaration node refers to its own copy of the reference variable.

CPN ML Reference

The colorset of the expression determines the colorset of the instance reference variable, which in turn determines the initial value.

Chapter 36

Expressions

In this chapter we first talk about the expression syntax, then expression evaluation, and finally expression use as inscription patterns and constructors.

Multiset expressions are not discussed in this chapter but are described, along with those predefined functions that operate on multisets, in Chapter 37, "Multisets."

Functional expressions are not discussed in this chapter but are described in Chapter 38, "Functions," along with the **let** and **case** forms.

Term Definitions

An *expression* is a sequence of values, CPN variables, type constructors, and operators that can be evaluated. The result of expression evaluation depends on both the types of operators and the CP net use of the expression.

A *canonical expression* is an expression that has itself as a value; it has been reduced to its lowest, or most fundamental form.

A *simple expression* is an expression whose components are all the same colorset.

Compound expressions are expressions whose components are simple expressions of different colorsets or other compound expressions.

Inscription expressions are expressions used as CP net arc inscriptions and guards. They do not contain reserved words, keywords, or semicolons, with the exception of

```
( ) [ ] { } , : = andalso orelse not  
if/then/else let/in/end case/of fn
```

Expression Syntax

Syntactic Specifiers and Reserved Words

The syntactic specifiers and reserved words that may be used in inscription expressions include:

```
( ) [ ] { } , : = andalso orelse
if/then/else let/in/end case/of fn
```

Their meanings are:

Specifier	Meaning	Where Described
()	tuple constructor expression associator	
[]	list constructor	
{}	record constructor	
,	tuple, list, or record component, or subexpression separator	
:		
=	equality operator	
andalso	boolean conjunction	
orelse	boolean disjunction	
if/then/else	boolean conditional	
let/in/end	local	
case/of	case	
fn	function	

Variables

Expressions used in code segments may contain CPN variables and reference variables, but inscriptions may contain only CPN variables.

Values

Val-defined identifiers may be anything legally definable by a `val` construct, including functions. See Chapter 33, “Values,” for a complete discussion of the `val` construct, and Chapter 38, “Functions,” for a discussion of val-defined identifiers whose value is a function.

Constructors

Constructors include:

- Tuple constructors:

`(id1, ..., idn)`

- Record constructors:

`{id1name = id1, ..., idnname = idn}`

- List constructors and operators:

`[id1, ..., idn]`

Chapter 32, “Colorsets,” provides complete descriptions of the constructors and their use.

Operators

Operators include:

- Arithmetic:

`~ * / div mod + -`

- String:

`^`

- List:

`:: ^^`

- Relational:

`= <> < > <= >=`

- Boolean:

`andalso orelse % \`

- Selector

`#`

Chapter 32, “Colorsets,” provides complete descriptions of the operators and their use.

Expression Evaluation

Expressions are evaluated within the context of their CP net location. The result of the evaluation depends on:

- 1) Component colorsets
- 2) Operator precedence
- 3) Associativity rules
- 4) CP net context

Evaluation Order

The default evaluation proceeds from left to right under the control of operator precedence and operator association.

The default evaluation order can be changed by the use of parenthesis grouping. For example, multiplication has a higher precedence than addition, so that evaluation of the following expression:

```
ML> 3 + 4*5;
```

yields:

```
> 23 : int
```

but

```
ML> (3 + 4)*5;
```

yields:

```
> 35 : int
```

Operator Precedence

Precedence is the default order in which different operators are evaluated. The Operator Precedence and Association Table below gives the default precedence for all the operators that are used in CPN ML expressions.

Association

Association is the direction of evaluation. An expression containing sequential occurrences of the same operator can be interpreted in several ways. For example, the expression:

5 - 4 - 3

can be interpreted as

(5-4) -3

or as

5 - (4 - 3).

In the first case the minus associates to the left and in the second the minus associates to the left.

The default association for operators of the same precedence is from left to right (left association).

Binary operators

All standard binary operators associate to the left.

Function Type Operator

The function type operator associates to the right. For example, the type:

`int -> real -> bool`

is to be interpreted as the type:

`int -> (real -> bool)`

and not as

`(int -> real) -> bool.`

Operator Precedence and Association Table

Operator	Meaning	Precedence	Association	Inscription
~	arithmetic negation	1	right	yes
*	arithmetic multiplication	7	left	yes
/	arithmetic real division	7	left	yes
div	arithmetic integer division	7	left	yes
mod	arithmetic modulo	7	left	yes
^	string concatenation	6	left	yes
+	arithmetic addition	6	left	yes
-	arithmetic subtraction	6	left	yes
::	list constructor	5	right	yes
^^	list concatenation	5	left	yes
=	equality	4	left	yes
<>	relational not equal	4	left	yes

CPN ML Reference

<	relational less than	4	left	yes
>	relational greater than	4	left	yes
<=	relational less than or equal to	4	left	yes
>=	relational greater than or equal to	4	left	yes
:=	reference access	3	left	no
o	function composition	3	left	no
andalso	boolean conjunction	2	left	yes
%	boolean selector			
\	boolean selector			
#	selector			
orelse	boolean disjunction	2	left	yes
case/of			left	yes
!	reference access			no
->	function type		right	no

Expression Uses

Patterns

Frequently the elements of a compound data object must be accessed independently of one another. Often the most convenient way is to set variables to the values of the separate elements, then use the variables as needed.

ML facilitates this type of access via pattern matching. A *pattern* is an ML expression that consists only of CPN variables and type constructors. It is used in input arc inscriptions to access and bind components of compound CPN data objects.

For example, given the following colorset and variable declarations:

```
color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

color No_of_Branches = int with 0..10;

color BankImportance = product
  No_of_Branches * Banks;

var national_banks,
    regional_banks, banks:Banks;

var branch_no:No_of_Branches;

var bank_size:BankImportance;
```

The following expressions are patterns:

```
(branch_no, regional_banks)
```

```
(branch_no, banks)
```

The following expression is not a pattern:

```
(9, Chase)
```

Tuple Pattern Matching

A tuple pattern is a sequence of variables that have the same type(s) as the elements in the tuple to be matched. The variables are enclosed in parentheses and separated by commas. For example:

```
(intvar, realvar, stringvar)
```

matches a three-element tuple in which the first element is an integer, the second is a real, and the third is a string.

List Pattern Matching

A pattern to match each element in a list is a sequence of variables of the type characteristic of the list. There must be as many variables as there are list elements. The variables are enclosed in brackets and separated by commas. For example:

```
[intvar1, intvar2, intvar3, intvar4]
```

matches a four-member list of integers.

Often the length of a list is unimportant or unknown, or only the first member is of specific interest. The double-colon pattern can be used in such cases: For example, a list of integers could be matched with the pattern:

```
intvar::intlistvar
```

The first variable will be set to the first element of the list, and the second will be set to the rest of the list. If the list contained only one member, `intlistvar` would be set to the empty list.

Constructors

A constructor is an expression that when evaluated yields a compound data object. They are used in guards and output arc inscriptions to generate and bind output variables.

It is often useful to construct a compound object whose values are given by expressions. CPN ML accomplishes this through the use

of constructors. There are appropriate constructors for each compound data type.

Constructor expressions may include any combination of identifiers, operators, and type constructors. They are distinguished from other expressions in that they evaluate to a compound data object. Thus, a pattern such as (a, b, c) when used as an output arc inscription is a constructor, whereas an expression such as:

```
(abs (intvar1 + intvar2),
  stringvar ^^ "New York",
  realvar1 <= realvar2, 4.4)
```

is not a pattern, but is a tuple constructor.

Tuple Constructors

A tuple constructor is a sequence of expressions that evaluate to data objects of same types as the elements of the type of tuple to be constructed. The sequence is enclosed in parentheses and the expressions are separated by commas, as with tuples generally. For example:

```
(abs (intvar1 + intvar2),
  stringvar ^^ "New York",
  realvar1 <= realvar2, 4.4)
```

constructs a tuple whose elements are an integer, a string, a boolean, and a real.

List Constructors

A list constructor is a sequence of expressions that evaluate to the same colorset. The sequence is enclosed in square brackets and the expressions are separated by commas. For example:

```
[intvar1* intvar2, if boolvar1
  then intvar3
  else intvar4, 46]
```

constructs a list consisting of three integers.

Expression Evaluation in CP Nets

Arcs

Expressions used as input and output arc inscriptions yield a multi-set when evaluated.

Guards

Expressions used as guards yield a boolean value when evaluated. They may both constrain enablement and bind a CPN variable in at the same time.

Code Segments

Expressions used in code segments yield values depending on the colorset and operators of the expression components. The only restriction is that code segments must not rebind CPN variables bound on the input arcs or guards.

Further Discussion

See Chapter 40, “Inscriptions,” for a complete discussion and copious examples of their use.

Chapter 37

Multisets

Term Definitions

A *token* is a typed data object. Tokens are not called objects to avoid confusion with graphical objects.

A *multiset* is a collection of tokens. Unlike ordinary sets each token may have duplicate values, hence the name multiset.

Multiset Variables

A *multiset variable* is a CPN variable that yields a multiset when evaluated.

An ordinary CPN variable can have any value of the colorset for which the variable is defined. The value of a multiset variable is a multiset of the colorset for which the variable is defined. See Chapter 34, “CPN Variables,” for further information about CPN variables.

Syntax

The syntax for declaring a multiset variable is:

```
var id1, id2, ....., idn : colorset ms;
```

This is the same as an ordinary CPN variable declaration, except for the addition of **ms** after the colorset name.

Use

Multiset variables can be referenced in a code segment and used in output arc inscriptions. They can be used to read in multiset expressions from an input file. This expression can then be evaluated and set as the value of the variable.

Multiset Creation

The *multiset creation operator*, backquote (```), creates a multiset containing a given value, a given number of times.

Syntax

```
int` colorset-value;
```

Operator from: (int * colorset_value) -> multiset

The integer argument must be non-negative. If this is not the case, an exception is raised.

Use

The multiset operator combined with multiset addition and subtraction provides a succinct method for specifying multisets. For example:

```
2`a + 1`c + 4`f
```

is a multiset consisting of seven CPN variables: two of variable a, one of variable c, and four of variable f.

Multiset expressions

A *multiset expression* is an ML expression that when evaluated yields a multiset. Multiset expressions are used as arc inscriptions. Input arc inscriptions may consist of regular multiset expressions, including functions.

Syntax

Examples

Two integer tokens each of whose value is the integer 3:

```
2`3
```

One integer variable named

```
(1`intvar1 + 4`intvar2)
```

A tuple pattern

```
(staff_type, order_type)
```

Multiset Constants, Operations, and Functions

Constant Definitions

Empty Multiset

The **empty** constant constructs an empty multiset that is polymorphic, e.g. identical for all kinds of multisets.

```
empty;
```

```
Constant: -> "a ms
```

For example:

```
ML> empty;
```

```
> empty : "a ms
```

Full Multiset

The **colorset** constant defines a constant which contains exactly one occurrence of each value in **colorset** for the full multiset over **colorset**. The **colorset** must have been declared with **ms** and must be small.

```
colorset_name;
```

```
Constant: -> <colorset> ms
```

For example:

```
color small_int = int with 1..10 declare ms;
```

```
ML> small_int;
```

```
> !! ((1,1),!! ((1,2),!! ((1,3),!! ((1,4),
!! ((1,5),!! ((1,6),!! ((1,7),
!! ((1,8),!! ((1,9),
!! ((1,10),empty)))))))) : small_int ms
```

Addition, Subtraction, and Scalar Multiplication, of Multisets

Addition (+ operator)

```
"a ms + "a ms;
```

```
Operator from: ("a ms * "a ms)-> "a ms
```

For example:

```
ML> 3`5 + 5`5;
> !! ((8,5),empty) : int ms
ML> 6`5 + 3`6;
> !! ((3,6),!! ((6,5),empty)) : int ms
```

Subtraction (- operator)

```
"a ms - "a ms;
```

Operator from: ("a ms * "a ms)-> "a ms

The first multiset must be greater than or equal to the second. If this is not the case, an exception is raised.

The second multiset must be a subset of the first multiset. If this is not the case, an exception is raised.

For example:

```
ML> 6`5 - 3`6;
> Failure: CPN'IllegalMultiSetSubtr
ML> 6`5 - 3`5;
> !! ((3,5),empty) : int ms
```

Scalar Multiplication (* operator)

For scalar multiplication the integer must be non-negative.

```
int * "a ms;
```

Operator from: (int * "a ms)-> "a ms

For example:

```
ML> 3*2`5;
> !! ((6,5),empty) : int ms
ML> 4`5 - 3`5;
> !! ((1,5),empty) : int ms
```

Multiplication of Multisets

`mult`'colorset can only be declared for tuple and record colorsets. It constructs a multiset over colorset by multiplying a set of multisets (one for each base colorset).

```
mult'colorset (colorset1 ms, colorset2 ms,
....., colorsetn ms);
```

Function from: (colorset₁ ms * colorset₂ ms
* * colorset_n ms)-> colorset ms

For example:

```
color integer = int;
color realno = real;
color tupleno = product integer * realno;
```

```
ML> mult'tupleno
      ((1`3 + 4`6), (3`3.0 + 4`6.0));

> !! ((3,(3,3.0)),!! ((4,(3,6.0)),!!
      ((12,(6,3.0)),!!
      ((16,(6,6.0)),empty)))) : baz ms
```

Comparing Multisets

These operations compare two multisets.

Equality (== operator)

```
"a ms == "a ms;
```

Operator from: ("a ms * "a ms) -> bool

For example:

```
ML> 6`5 == 8`5;

> false : bool
```

Not Equal (<><> operator)

```
"a ms <><> "a ms;
```

Operator from: ("a ms * "a ms) -> bool

For example:

```
ML> 6`5 <><> 8`5;
```

```
> true : bool
```

Less Than or Equal (<= operator)

```
"a ms <= "a ms;
```

Operator from: ("a ms * "a ms) -> bool

For example:

```
ML> 8`5 <= 8`5
```

```
> true : bool
```

Less Than (<< operator)

```
"a ms << "a ms;
```

Operator from: ("a ms * "a ms) -> bool

For example:

```
ML> 8`5 << 8`5
```

```
> false : bool
```

```
ML> 6`5 << 8`5
```

```
> true : bool
```

Greater Than or Equal (>= operator)

```
"a ms >= "a ms;
```

Operator: ("a ms * "a ms) -> bool

For example:

```
6`5 >= 8`5;
```

```
> false : bool
```

Greater Than (>> operator)

```
"a ms >> "a ms;
```

Operator: ("a ms * "a ms) -> bool

For example:

```
ML> 6`5 >> 8`5;
> false : bool
```

Coefficient

cf returns the coefficient of a given value in a given multiset.

```
cf ("a, "a ms);
```

Function from: ("a * "a ms)-> int

For example:

```
ML> cf (3, (5`6 + 3`6));
> 0 : int
```

Size

size returns the number of elements in a multiset.

```
size ("a ms);
```

Function from: "a ms-> int

For example:

```
ML> size (3`6 + 4`7);
> 7 : int
```

Random Value From a Multiset

random returns a random value from a given multiset. The probability for the selection of a value is proportional to its coefficient in the multiset.

random cannot be applied in net inscriptions such as arc inscriptions and guards. It can, however, be applied in code segments.

```
random ("a ms);
```

Function from: "a ms -> "a

For example:

CPN ML Reference

```
ML> random (3`6 + 4`7);  
> 6 : int
```

Conversion Between Lists and Multisets

list_to_ms and **ms_to_list** provide the conversion between the two types.

```
list_to_ms ["a, "b, ..., "c];
```

Function from: "a list -> "a ms

For example:

```
ML> list_to_ms [1,3,4];  
> !! ((1,1),!! ((1,3),!! ((1,4),empty)))  
: int ms
```

```
ms_to_list ("a ms);
```

Function from: "a ms -> "a list

For example:

```
ML> ms_to_list (1`3 + 4`6);  
> [6,6,6,6,3] : int list
```

Filter Function

filter constructs a filter (a function mapping multisets into smaller or identical multisets) from a predicate (a function mapping values into booleans).

```
filter ('a bool);
```

Function from: ('a -> bool) -> ("a ms -> "a ms)

Linear Extension of Functions

ext_col and **ext_ms** make a *linear extension* of a function.

```
ext_col ('a ''b);
```

Function from: ('a -> "b) -> ("a ms -> "b ms)

```
ext_ms ('a ' 'b ms);
```

Function from: ('a -> "b ms) -> ('a ms -> "b ms)

Multiset Input and Output

input_ms and **output_ms** allow code segments to read and write multisets.

```
input_ms (instream);
```

Function from: instream -> colorset ms

```
output_ms (outstream, <colorset> ms);
```

Function from: (outstream * colorset ms)-> unit

Internal Representation of Multisets

Design/CPN represents each multiset as a nested sequence of pairs, separated by double exclamation points (!!) where each pair contains a coefficient followed by a value. The constant `empty` denotes the empty multiset and terminates the sequence.

For example:

```
ML> 2`3 + 1`4 + 1`5 + 3`6;
> !! ((3,6),!! ((1,5),!! ((1,4),
    !! ((2,3),empty)))) : int ms
```

A multiset representation is said to be:

- Compressed, if each color occurs in at most one pair, with a positive coefficient.
- Sorted, if the colors occur in increasing order with respect to the ordering of the colorset.
- Normalized, if it is both compressed and sorted.

Design/CPN assumes all multiset representations to be compressed, but it does not assume them to be sorted.

When you apply the predeclared constants, operations, and functions, all multiset representations automatically become compressed.

When you define your own multisets or functions (between multisets) — by means of the internal representation of multisets — you *must* define them in such a way that all multiset representations become compressed. An easy way to do this is to apply the predeclared **compress** function.

Construct, Compress, and Sort

The following operations/functions are only necessary for users who want to work with the internal representation of multisets. The **!!** operation allows you to construct — and pattern match — the internal representation of multisets. The **compress** function allows you to compress a multiset representation.

Multiset constructor (!! operator)

```
!! ((int * "a) * "a ms);
```

Operator from : ((int * "a) * "a ms) -> "a ms

```
compress "a ms
```

Function from: "a ms -> "a ms

Timed Multisets

To support simulation with time, a new colorset is declared, when a time declaration is given:

```
colorset "a tms = !!! of
```

```
((int * "a * (TIMElist)) * ("a tms)) | empty;
```

Furthermore, **+**, **,**, **-**, **size**, **cf** and **compress** are overloaded to support timed multisets, when a time declaration is given.

Chapter 38

Functions

Declarations

Syntax

The syntax is:

```
fun identifier (parameter-tuple)
    = function-body;
```

identifier is associated with the **function-body**.

Datatype

The datatype of a function is the combined datatype of the domain (parameter) and the range (result). It is indicated by the reserved word **fn** followed by an expression whose syntax is:

```
fn parameter-type -> result-type
```

The function operator, **->**, indicates the mapping between the domain and range of the function.

Example

The following example increments its parameter by 1:

```
fun sum1 (n) = n + 1;
```

Local Declarations - Let

The **let** construct permits the declaration of locally-scoped variables within a function definition. In addition, **let** may be used within a code segment. This use provides the only case in CPN ML where **val** may be used outside of a declaration node.

Syntax

```
let val_declaration in expression end;
```

where `val_declaration` is a value declaration using `val`, and `expression`, is a CPN ML expression. See Chapter 33, “Values,” for a discussion of `val`, and Chapter 36, “Expressions,” for a discussion of CPN ML expressions.

Control Structures

if-then-else

if-then-else provides a conditional control structure that may be used in function declarations, inscriptions, and code segments.

Syntax

```
if boolean-expression
  then expression
  else expression;
```

The uses of the if-then-else construct determine which expressions are legal in the then and else clauses:

Input Arc Inscription: CPN variables may not be bound by a conditional input arc inscriptions; they must be bound somewhere else on the transition - either on a different input arc or in the guard.

Guard: Each expression must be a boolean expression, evaluating to true or false. Thus the following example is legal because each variable binding is either true or not true:

```
[if a = 3 then b = 4 else c = 5]
```

The following example is not legal because the then and else clauses evaluate to integers:

```
[if a = 3 then 4 else 5]
```

Output Arc Inscription: CPN variables may be bound by a conditional output arc inscription.

Boolean Selectors

These operations allow you to apply a *boolean selector* to a value, a multiset, a pair of values or a pair of multisets. The first two op-

erations work as an if-then command, while the last two work as an if-then-else.

```
% (bool list * 'a) -> 'a ms
/ (bool list * 'a ms) -> 'a ms
```

Case

The `case` structure is a conditional expression generators that provides an if-then-else, if-then-else,... capability.

Syntax

```
case identifier of
  boolean_test1 |
  boolean_test2 |
  ...
end;
```

The evaluation proceeds until there is a match. If none of the tests succeed then the result of evaluation is false.

Function Invocation

To use a function, apply it to its argument, which must be in the domain of the function. For example, to add 1 to the number 100: **sum1**to the argument 100:

```
ML> sum1 (100);
> 101 : int
```


Chapter 39

Timing

Time Stamps

For simulation with time, each token may, in addition to its token value, carry a *time stamp*, that is, an integer or real expression. The time stamps are calculated in the CPN Simulator by adding the transition delay, if any, and the arc delay, if any, to the model time at which the token is created.

Declaration

Each colorset that is to be timed must append the keyword `timed` to the colorset declaration. For example:

```
color A = with id1 | id2 | id3 | id4 timed;
```

Defaults for Time Stamps

Index, enumeration, and simple colorsets are untimed by default.

Compound colorsets (that is, unions, products, records, and lists) are time if and only if at least one of the base colorsets is timed. In the case where more than one base colorset is timed, the compound colorset will still only have a single time stamp for each token.

Duplicate and subset colorsets are timed by default if and only if the base colorset is timed.

To make a default timed colorset non-timed, append the keyword **untimed** to the colorset declaration. For example:

```
color colorset = colorset0 untimed;
```

Timed Arc Inscriptions

Input Arc Inscriptions

To ignore the time stamps of tokens from an input place whose colorset is timed, append the keywords `@ignore` to the input arc inscription. For example:

The following declaration creates a timed colorset:

```
color Banks = with HarvardTrust |  
              Shawmut | NationalGrand timed;
```

so that an input place with `Banks` as a colorset is timed. The following input arc inscription:

```
2`Banks@ignore
```

causes the simulator to ignore that specification.

Input arc inscriptions may not specify time delays.

Output Arc Inscriptions

Output arc inscriptions may specify time delays by appending a time delay expression to the output arc inscription. The time delay expression consist of the keyword `@+` followed by an integer or real expression. For example, the following output arc inscription:

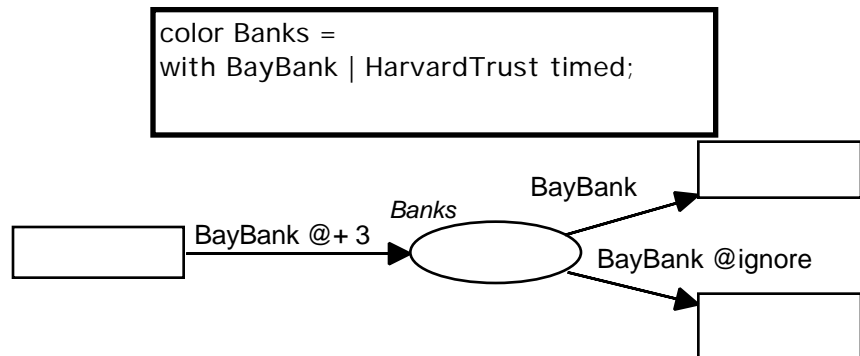
```
2`Banks @+5
```

causes the simulator to put a time delay on the tokens going into the output place. A missing arc delay is equivalent to a zero time delay.

Output arc delays can use CPN variables and the function `time`.

Combined Use

The CP net below shows an example of how the input and output arc time delays can interact:

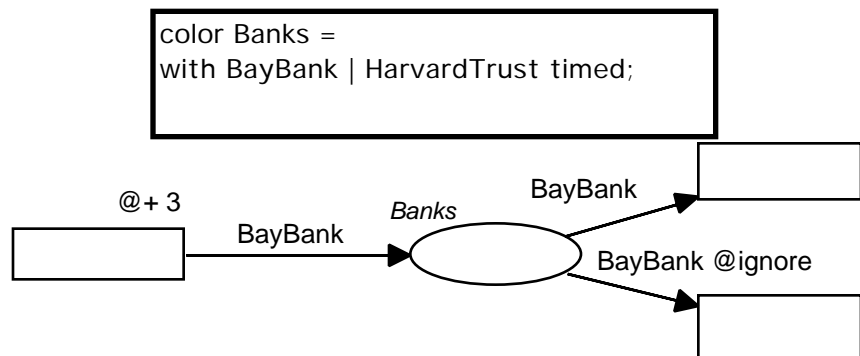


Time region

The time region is a region of a transition that specifies a time delay. Transitions may specify time delays by specifying an integer or real expression preceded by @+. For example, the following transition time delay:

@+5

causes the simulator to put a time delay of + 5 time units on all tokens leaving the transition. The example CPN net above could be expressed with a transition time region instead of a timed output arc inscription:



A missing arc delay is equivalent to a zero time delay.

Time regions can use all the CPN variables of the corresponding transition. This means that the time delay may depend upon the token values for input and output tokens.

Moreover, the expression may, via calculated CPN variables, depend upon reference variables and input files. (See the discussion of calculated CPN variables in the code segment discussion in Chapter 40, "Inscriptions.")

Finally, the time region is also allowed to use `time()`.

Time-Related Functions

Append time list to one-element multiset (@ operator)

The @ operator takes a one-value multiset and appends a list of time stamps. The length of the list must be equal to the number of occurrences of the value in the multiset. The function can only be used if time has been declared. If the multiset has more than one value, an exception is raised.

```
multiset_specification @
  [integer_list_specification];
```

Function from: ("a ms * int list) -> "a tms

For example:

```
ML> 2`6 @[1,2];
> !!! ((2,6,[1,2]),empty) : int tms
```

Time and step

The current model time and the step number can be inspected by means of the functions:

```
time ();
```

Function from: unit -> integer or real

```
step ();
```

Function from: unit -> integer

Time and **step** can be used in initial marking, arc inscription, guard, code and time regions, with the following exceptions: **Time** cannot be used in input arc inscriptions, guards and code guards, while **step** may only be used for reporting purposes in code segment action clauses.

Simulation options

with_time tests whether or not the simulation option *Simulation with time* is currently in force.

```
with_time ();
```

Function from: unit -> bool

Chapter 40

Inscriptions

This section describes the inscriptions associated with CPN net components. An *inscription* is a CPN ML construct that affects the behavior of the net. Inscriptions vary in their syntactic requirements depending on the type of inscriptions. The inscription types include:

- Colorset region (place)
- Initial marking region (place)
- Arc inscription region (arcs)
- Time region (transition)
- Guard region (transition)
- Code segment (transition)

We present the inscriptions in the context of a sample model in which the regions associated with each place, arc, and transition are shown and briefly described. Each inscription is defined along with its syntax and one or more examples.

Example Conventions

In the examples in this chapter we use the default typographic conventions used by CPN:

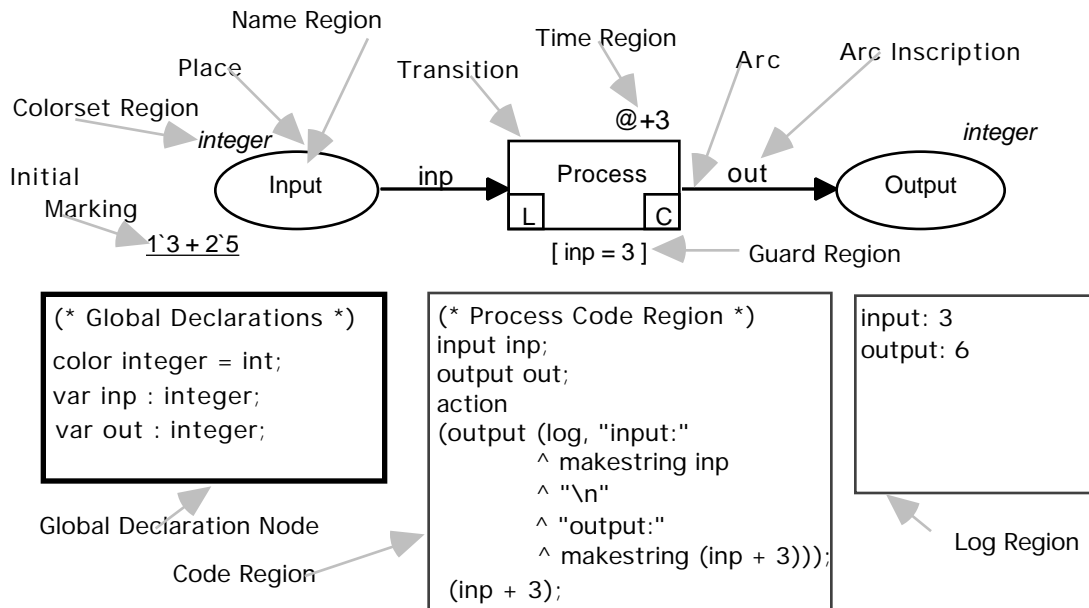
- Initial markings are underlined
- Colorset regions are italic

In addition, guards are always enclosed in square brackets to distinguish them from other inscriptions. Square brackets are only required for guards when they consist of a list of expressions.

CPN ML Reference

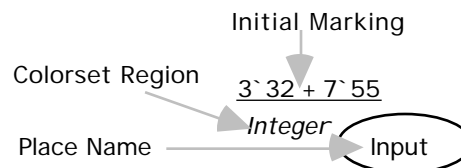
Example Diagram

The diagram below has an example of each inscription and region we will discuss in this chapter. The global declaration node is not an inscription, but is included because it is an essential part of any model.



Places

There are three regions that may be associated with a place. Two are optional and one is required:



- Name region - optional
- Colorset region - required
- Initial marking region - optional

Place Name Region

The name region is a label that identifies the place. It is not a CPN ML inscription and may contain any sequence of characters. The

Syntax Options dialog (**Set** menu) provides the additional validation criteria:



Colorset Region

The colorset region contains the identifier of a colorset declared in a global or temporary declaration node. The colorset determines the colorset of all the tokens that can be put in the place.

Initial Marking Region

The *initial marking* is a multiset expression that specifies the initial tokens for a place. The colorset for the expression must match the colorset of the place. If the colorset is timed, the initial marking can specify a time delay.

The initial marking is optional. When an initial marking is absent, it is equivalent to empty, the empty multiset.

Optional Time Delays

An initial marking time delay is an expression of type TIME that is appended to the initial marking expression with @+ as a separator. A missing initial marking time delay is equivalent to a zero delay.

When an initial state is generated, each timed token gets a time stamp which is equal to the start value of model time plus the initial marking delay of the corresponding place. This ensures that all the initial tokens get the same time stamp.

Syntax

The initial marking region has the form:

```
initial-marking <<@+ init-mark-delay>>
```

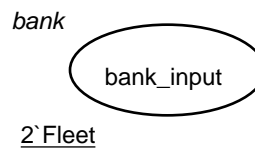
CPN ML Reference

where *initial-marking* is a multiset expression and *init-mark-delay* is an expression whose SML datatype is integer or real.

Examples

Example 1: In the example below *bank*, the colorset for the place *bank_input*, is declared:

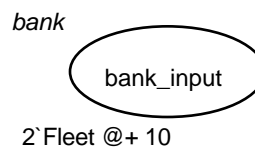
```
color bank =  
    with Fleet | BayBank | Shawmut | NationalGrand;
```



2`Fleet specifies two initial tokens each of whose value is the constant *Fleet*.

Example 2: In the example below *bank*, the colorset for the place *bank_input*, is declared:

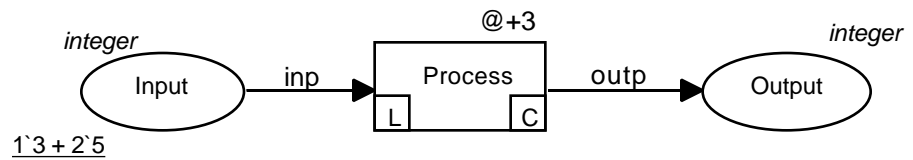
```
color bank =  
    with Fleet | BayBank | Shawmut | NationalGrand  
    timed;
```



2`Fleet @+ 10 specifies two initial tokens each of whose value is the constant *Fleet*. @+10 specifies a time delay of 10 time units.

Arcs

Arcs have one region associated with them - the arc inscription region. The following figure shows two arc inscriptions - one input arc inscription and one output arc inscription. Each arc inscription has a CPN variable, *inp*, of colorset integer.



Arc Inscription Region

An arc inscription is a CPN ML expression that evaluates to a multiset.

A missing arc inscription is equivalent to `empty (tempty)`, the empty (timed) multiset.

Patterns and Constructors

See Chapter 36, “Expressions,” for a discussion of patterns and constructors as arc inscriptions.

Side Effects

Arc inscriptions are *not* allowed to have side effects and cannot:

- Contain input/output or `use` commands.
- Update or use reference variables.
- Use the `random` function.

This restriction is currently not checked by Design/CPN and the system may malfunction if the restriction is violated.

Free variables are output arc variables that have not been bound on an input arc or in the guard. They are assigned random values when executing in interactive mode.

Caution: Do not use free variables when executing in automatic mode, since that will stop the simulator.

Time Stamps

An input arc inscription can override a time stamp on an input place by specifying `ignore`. The syntax is:

```
arc-inscription@ignore.
```

Specifying `ignore` causes time stamps not to be taken into account when the time enabling of the transition is calculated.

An output arc delay must be an expression of type TIME and it must be appended to an output arc inscription, using @+ as a separator. The syntax is:

```
arc-exp @+arc-delay.
```

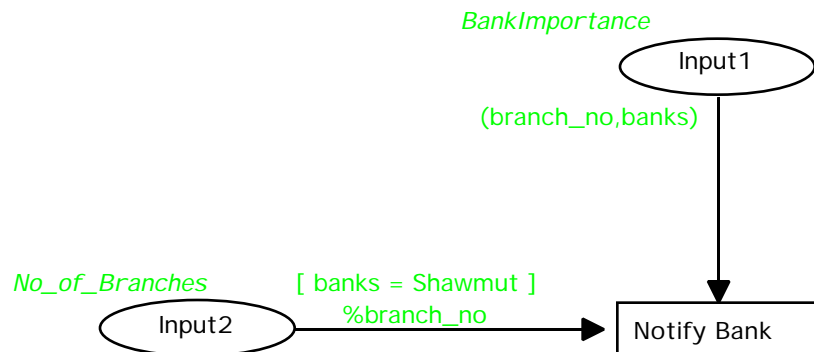
An omitted output arc delay is equivalent to a zero delay.

Arc delays can, in exactly the same way as transition delays, use CPN variables and `time()`.

Conditional arc inscriptions

Conditional arc inscriptions may be used on input arcs, if all the variables are bound in the guard or in another input arc.

In the example below, the input arc inscription from `Input2` can use conditional boolean selectors (`%`, which means "if then") because the variables `branch_no` and `banks` are bound on the input arc from `Input1`. See Chapter 38, "Functions," for a discussion of boolean selectors and the `if-then-else` structure.



Conditional arc inscriptions on output arcs may bind output variables mentioned in the output clause of the code segment.

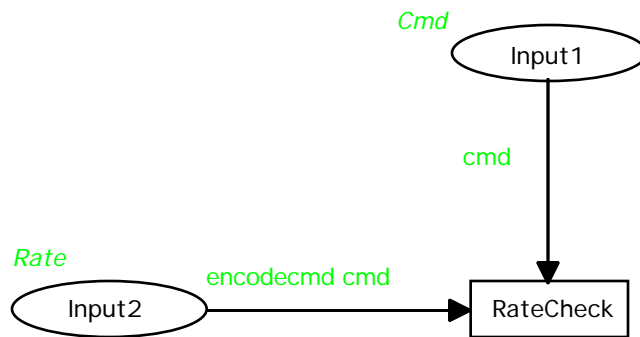
Function application

Functions used in arc inscriptions must not have side effects.

Functions may be used on input arcs only if the function parameters are bound elsewhere at that transition.

Functions on output arc inscriptions may bind output variables.

In the example below, the input arc inscription from `Input2` can use a function because the variable `cmd` is bound on the input arc from `Input1`:



Input Arc Inscription Examples - Declarations

Colorset Declarations

```

(* Enumerated colorset *)

color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

(* Indexed colorset *)

color BankBranch =
  index branch_number with 1...8;

(* Constrained integer colorsets *)

color No_of_Branches = int with 0..10;

color No_of_Customers = int with 0..99999;

(* Tuple colorset *)

color BankImportance = product
  No_of_Branches * Banks;

(* Record colorset *)

color BankImportanceRec = record
  Size:No_of_Branches *
  Name:Banks;

(* Union colorset *)

color RegionBanksSize = union Name:Banks +
  CustomerBase : No_of_Customers +
  BranchNumbers : No_of_Branches;

(* List colorset *)

color BankList = list Banks;
  
```

CPN ML Reference

```
(* Subset colorset *)  
  
color LocalBanks = subset Banks  
    with [NationalGrand,  
         MarbleheadSavings];
```

Variable declarations

```
(* enumerated colorset *)  
  
var national_banks,  
    regional_banks, banks:Banks;  
  
(* indexed colorset *)  
  
var branch:Bank_Branch;  
  
(* integer colorset *)  
  
var branch_no:No_of_Branches;  
  
(* integer colorset *)  
  
var customers : No_of_Customers;  
  
(* tuple colorset *)  
  
var bank_size:BankImportance;  
  
(* record colorset *)  
  
var bank_size_rec :BankImportanceRec;  
  
(* union colorset *)  
  
var regionbank:RegionBanksSize;  
  
(* list colorset *)  
  
var banklist:BankList;  
  
(* subset colorset *)  
  
var local_bank : LocalBanks;
```

Input Arc Inscription Examples

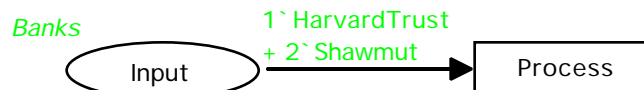
Constant:

```
color Banks = with HarvardTrust | Chase  
    | MarbleheadSavings | Shawmut  
    | NationalGrand;
```



Multiset expression with enumerated values:

```
color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;
```



Variable:

```
color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;
```

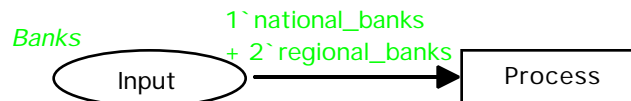
```
var banks:Banks;
```



Multiset expression with variables:

```
color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;
```

```
var national_banks,
    regional_banks, banks:Banks;
```

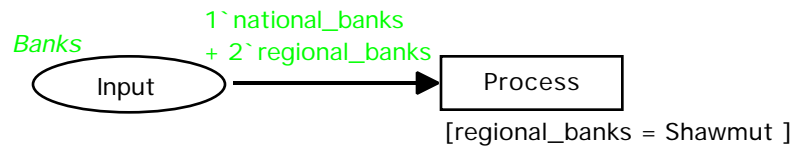


Multiset expression using variables constrained by

a guard:

```
color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;
```

```
var national_banks,
    regional_banks, banks:Banks;
```

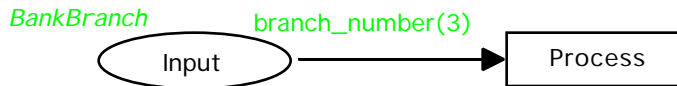


Indexed values:

```

color BankBranch =
    index branch_number with 1...8;

var branch_no:No_of_Branches;
  
```



Tuple colorset, CPN variable arc inscription:

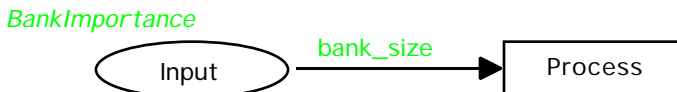
```

color No_of_Branches = int with 0..10;

color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

color BankImportance = product
    No_of_Branches * Banks;

var bank_size:BankImportance;
  
```



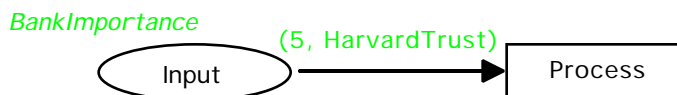
Tuple colorset, pattern arc inscription:

```

color No_of_Branches = int with 0..10;

color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

color BankImportance = product
    No_of_Branches * Banks;
  
```



Record colorset, pattern arc inscription with variables:

```

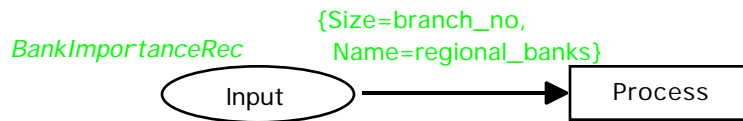
color No_of_Branches = int with 0..10;

color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

color BankImportanceRec = record
    Size:No_of_Branches *
    Name:Banks;

var branch_no:No_of_Branches;

var regional_banks:Banks;
    
```



List colorset, list constructor arc inscription:

```

color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

color BankList = list Banks;

var banks:Banks;

var bank_list:BankList;
    
```



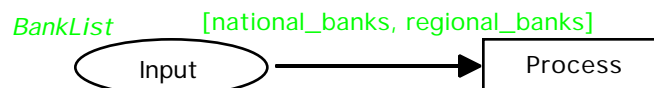
List colorset, list pattern arc inscription:

```

color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

color BankList = list Banks;

var national_banks,
    regional_banks:Banks;
    
```



Union colorset:

In this example, the union colorset `RegionBanksSize` is used in an input place that sends different components of the union to different transitions for different kinds of processing. The output places of those transitions have colorsets corresponding to the inputs that came into the transitions.

The colorset of the first place, `Output1`, is simply the colorset of one of the union colorset components - `No_of_Customers`. The colorset of the second place, `Output2`, is a tuple, `BankImportance`, that combines two of the union colorset components - `No_of_Branches`, and `Banks`.

```
color Banks = with HarvardTrust | Chase
              | MarbleheadSavings | Shawmut
              | NationalGrand;

color No_of_Customers = int with 0..99999;

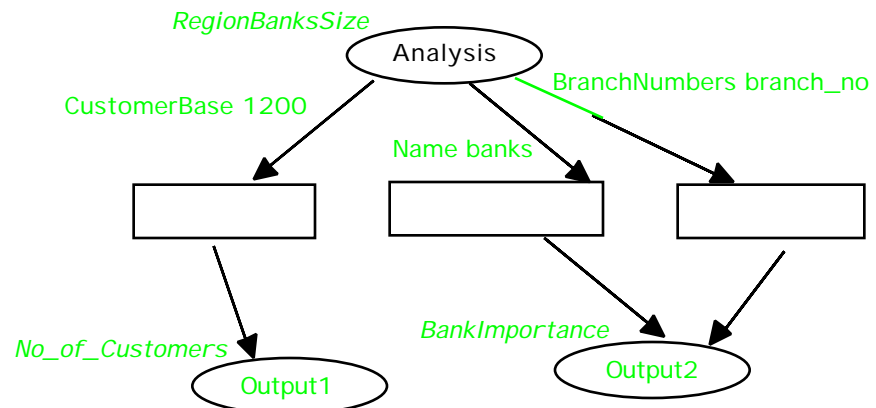
color No_of_Branches = int with 0..10;

color BankImportance = product
  No_of_Branches * Banks;

color RegionBanksSize = union Name:Banks +
  CustomerBase : No_of_Customers +
  BranchNumbers : No_of_Branches;

var banks:Banks;

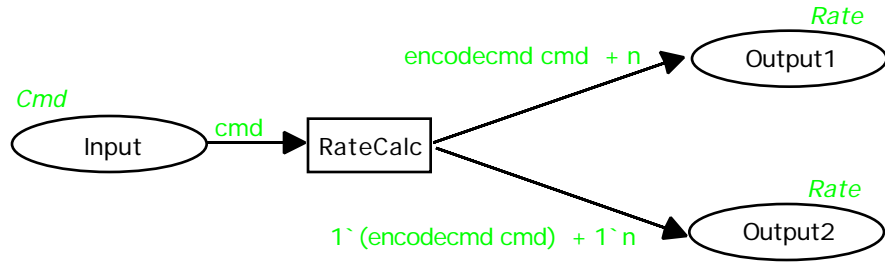
var branch_no:No_of_Branches;
```



Output Arc Inscription Examples

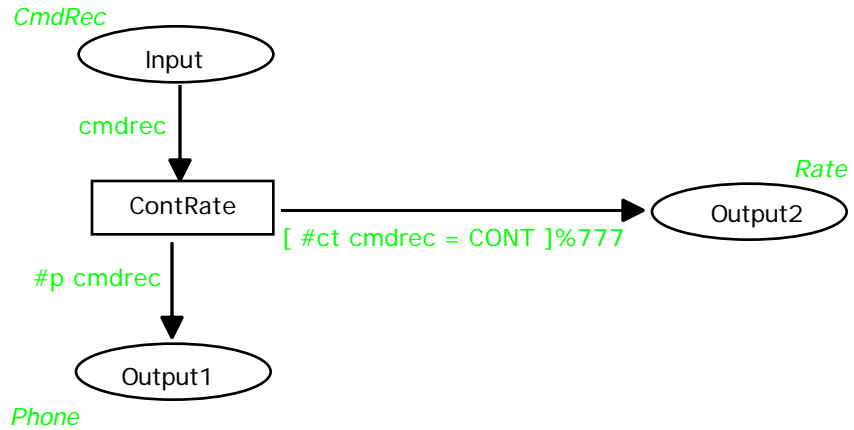
Function application

In this example the first output arc has one token, the second output arc has two. Note the use of free variables.

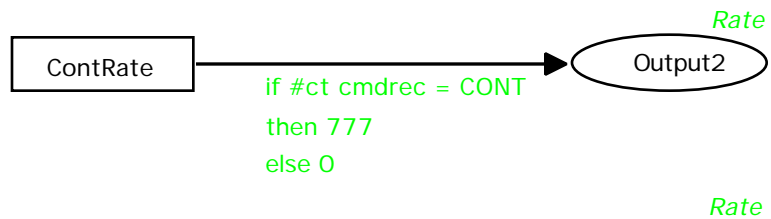


Conditional output arc inscriptions

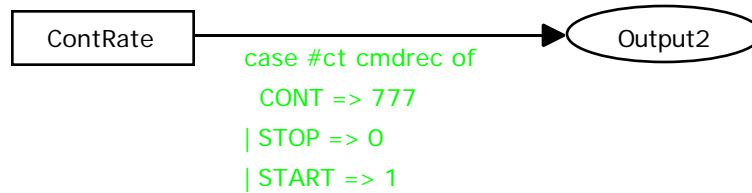
Example 1



Example 2



Example 3

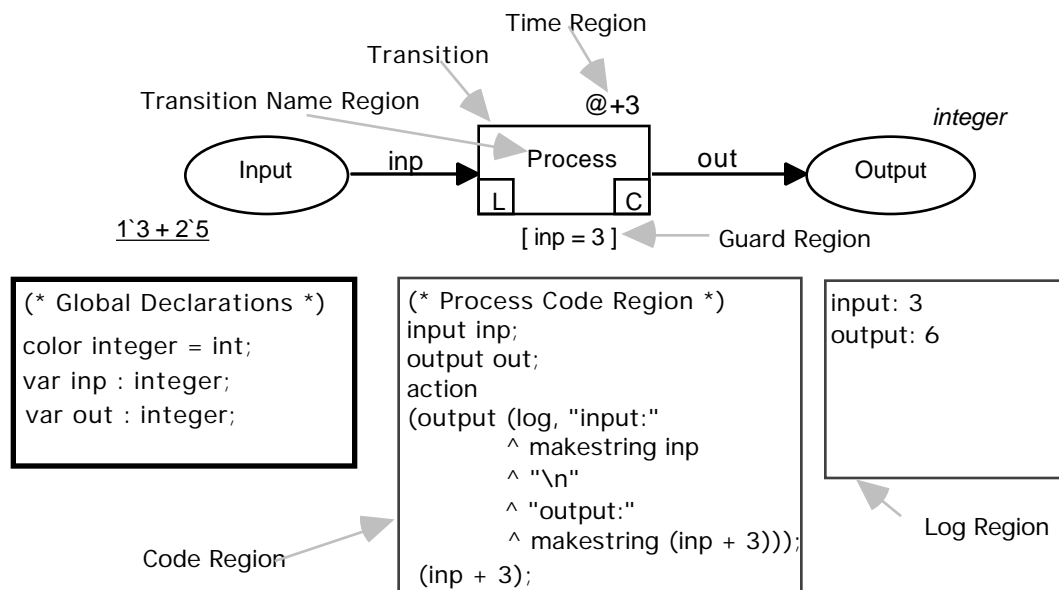


Unbound variables on an output arc



Transition Regions

There are five regions that may be associated with a transition. All are optional:



Transition Name Region

The name region is a label that identifies the transition. It is not a CPN ML inscription and may contain any sequence of characters. The **Syntax Options** dialog (**Set** menu) provides the additional validation criteria.

Time Region

A transition delay must be a positive integer or real expression. The expression is preceded by @+ and this means that the time region has the form @+ delay-expr .

Time delay is always added relative to the current time. for example, if current time is 10 and the time delay is @+2 then the time stamp of tokens sent to the output place will be 12.

A missing time region is equivalent to a zero delay.

The time delay expression can use all the CPN variables of the corresponding transition. This means that the time delay may depend upon the token values for input and output tokens.

Moreover, the expression may, via calculated CPN variables, depend upon reference variables and input files.

The time region may use the random function and the time function.

Guard Region

An guard is a CPN ML boolean expression that evaluates to a true or false.

Syntax:

- a boolean expression or
- a list of boolean expressions: [b-exp₁, b-exp₂, ...]

Use

Guards are used for *tests* on input arc inscription variables (enabling restrictions), typically with the following syntactical elements:

```
=    <>    <=    >=    <    >
andalso    orelse
```

Guards are also used to restrict values of output arc inscription variables, based on the values of the input arc inscription variables

Side Effects

Guards are *not* allowed to have side effects and cannot:

- Contain input/output or use commands.
- Update or use reference variables.
- Use the `random` function.

This restriction is currently *not* checked by Design/CPN and the system may malfunction if the restriction is violated.

Conditional Expressions

Each expression used in a guard must be a boolean expression, evaluating to true or false. Thus the following example is legal because each variable binding is either true or not true:

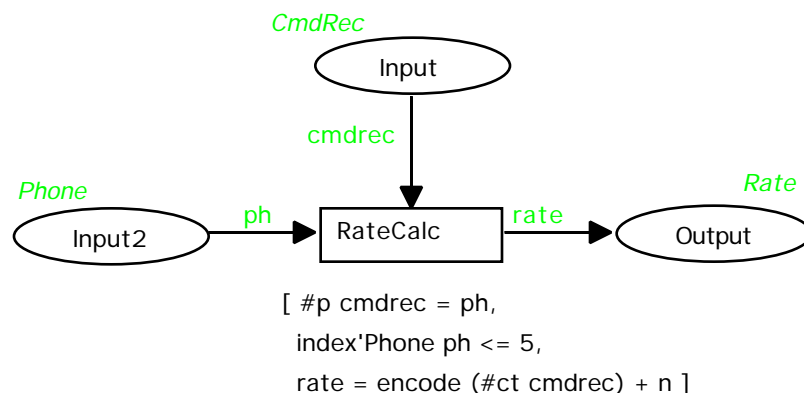
```
[if a = 3 then b = 4 else c = 5]
```

The following example is not legal because the `then` and `else` clauses evaluate to integers:

```
[if a = 3 then 4 else 5]
```

Example

A transition and a guard with two tests, providing a value to the output variable `rate`.



Code Segment

Each transition may have a code segment, which is a sequential piece of CPN ML code, executed each time the transition occurs.

Typical uses of transition code segments:

- Instrumentation (for example, statistics, tests, reports)
- Communication (for example, with the user through dialogs and the files)
- Complex calculations
- Graphic animation

CPN Variable Restrictions

Code segments may use CPN variables that appear on the input arc and in the guard. However, code segments may not change the value of input arc and guard variables since the input arc and guard variables are already bound by the time the code segment is executed.

Code segments may calculate the value of CPN variables that appear on the output arc since these are not bound at the time the code segment is executed.

Syntax

The code segments have syntax restrictions. Each code segment may contain an ordered list of one each of the following:

- Input clause (optional)
- Output clause (optional)
- Code action clause (mandatory)

Input Clause

The input clause is identified by the keyword `input` and is a CPN variable or a tuple of CPN variables without repetitions.

The input pattern lists the CPN variables that can be used in the code action clause. If the input clause is omitted, it implies that no CPN variables can be applied in the code action.

As described above, under **CPN Variable Restrictions**, the code action can apply the values of these CPN variables but it cannot change them.

The CPN variables listed in the input pattern can be used in the code action even if you have declared an SML identifier with the same name in the declaration node.

Output Clause

The output clause is identified by the keyword `output` and is a CPN variable or a tuple of CPN variables without repetitions.

The output clause lists the CPN variables that are to be calculated in the code action clause; these variables are said to be calculated CPN variables. If the output clause is omitted, no CPN variables may be calculated.

The output clause is a tuple whose type is the cartesian product of the types of the variables. The result of evaluating the action expression is bound to the output variables in the order listed in the output clause.

Code Action Clause

The code action clause is identified by the keyword `action` and is a CPN ML expression.

The code action cannot contain any colorset, CPN variable, or reference variable declarations. However, new functions and constants can be defined for local use by means of the `let/in/end` construct, as shown in Example 1. See Chapter 38, "Functions," for a discussion of the `let/in/end` construct.

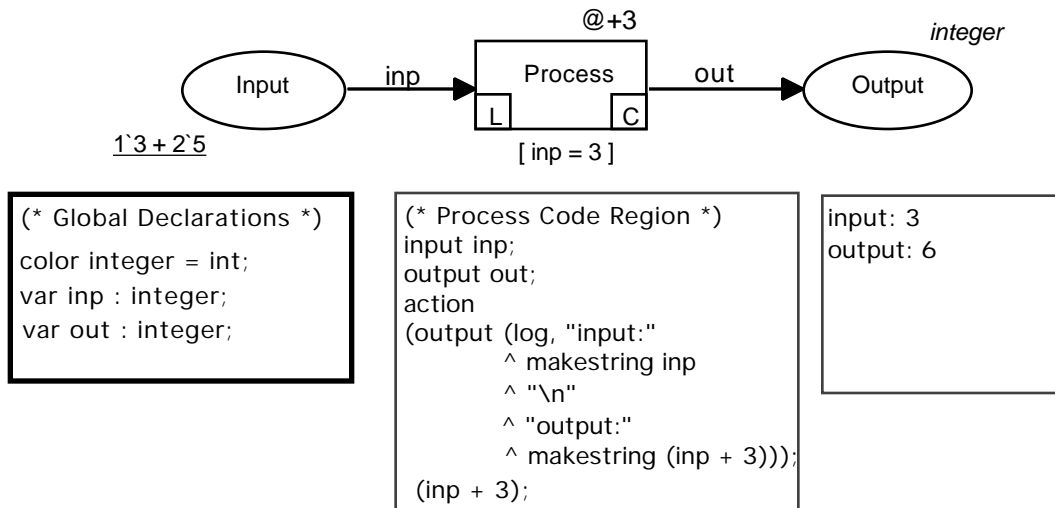
In addition, it can apply user-declared and predeclared constants, operations, and functions.

The code action is executed as a local declaration in an environment enriched by the input arc and guard CPN variables. This guarantees that the code action cannot directly change any CPN variables but only local copies of them. When the code action has been executed, its result is applied to bind the output arc CPN variables.

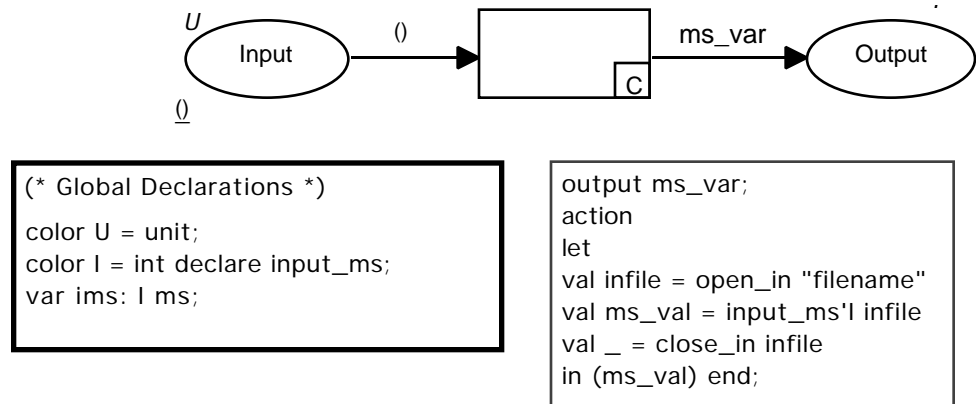
If no output clause is given, its type is assumed to be `unit`.

Examples

Example 1:



Example 2:



Chapter 41

The CPN ML Runtime Environment

Predeclared Environment

Design/CPN automatically generates a runtime environment containing a set of predeclared constants, operations and functions which you can apply to manipulate values, multisets and functions:

- A *constant* is an identifier with a fixed value.
- A *function* always takes one argument (at a time), and its name is written before the argument. The argument may be a tuple, for example: `dist 'A (c, f)`.
- An *operation* always takes exactly two arguments, and the name of the operation is written between the arguments, for example: `3 + 7`.

Predefined Constants, Functions, and Operators

Empty Multiset

The **empty** constant constructs an empty multiset that is polymorphic, e.g. identical for all kinds of multisets.

```
empty;
```

Constant: -> "a ms

Addition, Subtraction, and Scalar Multiplication, of Multisets

Addition (+ operator)

```
"a ms + "a ms;
```

Operator from: ("a ms * "a ms)-> "a ms

Subtraction (- operator)

```
"a ms - "a ms;
```

Operator from: ("a ms * "a ms)-> "a ms

The first multiset must be greater than or equal to the second. If this is not the case, an exception is raised.

The second multiset must be a subset of the first multiset. If this is not the case, an exception is raised.

Scalar Multiplication (* operator)

For scalar multiplication the integer must be non-negative.

```
int * "a ms;
```

Operator from: (int * "a ms)-> "a ms

Multiplication of Multisets

`mult'colorset` can only be declared for tuple and record colorsets. It constructs a multiset over `colorset` by multiplying a set of multisets (one for each base colorset).

```
mult'colorset (colorset1 ms, colorset2 ms,  
....., colorsetn ms);
```

Function from: (colorset₁ ms * colorset₂ ms
* * colorset_n ms)-> colorset ms

Comparing Multisets

These operations compare two multisets.

Equality (== operator)

```
"a ms == "a ms;
```

Operator from: ("a ms * "a ms) -> bool

Not Equal (<><> operator)

```
"a ms <><> "a ms;
```

Operator from: ("a ms * "a ms) -> bool

Less Than or Equal (<=> operator)

```
"a ms <= "a ms ;
```

Operator from: ("a ms * "a ms) -> bool

Less Than (<< operator)

```
"a ms << "a ms ;
```

Operator from: ("a ms * "a ms) -> bool

Greater Than or Equal (>=> operator)

```
"a ms >= "a ms ;
```

Operator: ("a ms * "a ms) -> bool

Greater Than (>> operator)

```
"a ms >> "a ms ;
```

Operator: ("a ms * "a ms) -> bool

Multiset Creation

The *multiset creation operator*, backquote (```), creates a multiset containing a given value, a given number of times.

```
int` colorset-value ;
```

Operator from: (int * colorset_value) -> multiset

The integer argument must be non-negative. If this is not the case, an exception is raised.

Coefficient

cf returns the coefficient of a given value in a given multiset.

```
cf ("a, "a ms) ;
```

Function from: ("a * "a ms) -> int

Size

size returns the number of elements in a multiset.

```
size ("a ms);
```

Function from: "a ms-> int

Random Value From a Multiset

random returns a random value from a given multiset. The probability for the selection of a value is proportional to its coefficient in the multiset.

random cannot be applied in net inscriptions such as arc inscriptions and guards. It can, however, be applied in code segments.

```
random ("a ms);
```

Function from: "a ms -> "a

Conversion Between Lists and Multisets

list_to_ms and **ms_to_list** provide the conversion between the two types.

```
list_to_ms ["a, "b, ..., "c];
```

Function from: "a list -> "a ms

Filter Function

filter constructs a filter (a function mapping multisets into smaller or identical multisets) from a predicate (a function mapping values into booleans).

```
filter ('a bool);
```

Function from: ('a -> bool) -> ("a ms -> "a ms)

Linear Extension of Functions

ext_col and **ext_ms** make a *linear extension* of a function.

```
ext_col ('a "b);
```

Function from: ('a -> "b) -> ("a ms -> "b ms)

```
ext_ms ('a "b ms);
```

Function from: ('a -> "b ms) -> ('a ms -> "b ms)

Multiset Input and Output

`input_ms` and `output_ms` allow code segments to read and write multisets.

```
input_ms (instream);
```

Function from: `instream -> colorset ms`

```
output_ms (outstream, <colorset> ms);
```

Function from: `(outstream * colorset ms)-> unit`

Append time list to one-element multiset (@ operator)

The @ operator takes a one-value multiset and appends a list of time stamps. The length of the list must be equal to the number of occurrences of the value in the multiset. The function can only be used if time has been declared. If the multiset has more than one value, an exception is raised.

```
multiset_specification @  
  [integer_list_specification];
```

Function from: `("a ms * int list) -> "a tms`

For example:

```
ML> 2^6 @[1,2];  
> !!! ((2,6,[1,2]),empty) : int tms
```

Time and step

The current model time and the step number can be inspected by means of the functions:

```
time ();
```

Function from: `unit -> integer or real`

```
step ();
```

Function from: `unit -> integer`

Time and **step** can be used in initial marking, arc inscription, guard, code and time regions, with the following exceptions: **Time** cannot be used in input arc inscriptions, guards and code guards, while **step** may only be used for reporting purposes in code segment action clauses.

Simulation options

with_time and **with_code** test whether or not the simulation options *Simulation with time* and *simulation with code* are currently in force.

```
with_time ();
```

Function from: unit -> bool

```
with_code ();
```

Function from: unit -> bool

These functions can be used in initial marking, output arc inscription, and code segments. They may not be used in input arc inscriptions and guards.

Simulation management

write_report manipulates the simulation reports given by CPN. It is possible to turn off the simulator's use of the facility and use the facility for other purposes.

```
write_report (string);
```

Function from: string -> unit

stop_simulation permits the simulation to be stopped from within a code segment. This provides a way of handling exceptions from within the model.

```
stop_simulation ();
```

Function from: unit -> unit

Boolean Selector

% and **/** apply a *boolean selector* to a value, a multiset, a pair of values or a pair of multisets.

% is an if-then command, while **/** is an if-then-else.

```
[boolean list] % 'a;
```

Operator from: (bool list * 'a) -> 'a ms

```
[boolean list] / 'a ms;
```

Operator from: (bool list * 'a ms) -> 'a ms

Addition, Subtraction, and Scalar Multiplication of Functions

Addition (+ operator)

```
("a -> "b ms) + ("a -> "b ms);
```

```
Operator from: (("a -> "b ms) *  
("a -> "b ms)) -> ("a -> "b ms)
```

Subtraction (- operator)

```
("a "b ms) - ("a "b ms);
```

```
Operator from: (("a -> "b ms) *  
("a -> "b ms)) -> ("a -> "b ms)
```

Function subtraction applies multiset subtraction.

The first multiset must be greater than or equal to the second. If this is not the case, an exception is raised.

The second multiset must be a subset of the first multiset. If this is not the case, an exception is raised.

Scalar Multiplication (* operator)

For scalar multiplication the integer must be non-negative.

```
(int * ('a 'b ms));
```

```
Operator from: ((int * ('a -> 'b ms)) ->  
('a -> 'b ms))
```

Identity, Zero, and Ignore

id, **zero**, and **ign** can be applied to all kinds of arguments, for example: values and multisets. They can even be applied to functions.

Identity

Id maps everything into itself.

```
id ("a);
```

```
Function from: 'a -> 'a
```

Zero

Zero maps everything into the empty multiset, which is polymorphic (that is, the same for all kinds of multisets).

```
zero ("a");
```

Function from: "a -> "b ms

Ignore

Ign maps everything into () — the only element in the primitive ML type unit.

```
ign ("a");
```

Function from: "a -> unit

Graphical Functions

You can use ML functions to access most of the functions in the graphic library of Design/OA. Users of Design/OA should note that the functions have the same names as their C counterparts.

Using these functions you can write animation code for code segments of transitions. The functions should only be used to manipulate auxiliary objects.

ColorIO Functions

These functions allow code segments to read and write colorset and multiset definitions.

```
input_col (instream);
```

Function from: instream -> <colorset>

```
output_col (instream, colorset);
```

Function from: (outstream * <colorset>) -> unit

```
input_ms (instream);
```

Function from: instream -> <colorset> ms

```
output_ms (outstream, colorset ms);
```

Function from: (outstream * <colorset> ms) -> unit

```
input_tms (instream);
```

Function from: `instream ->`
(`<colorset> ms * TIME list`)

```
output_tms (outstream, colorset tms);
```

Function from: `(outstream * <colorset> tms) -> unit`

NOTE: The timed multiset functions (`input_tms` and `output_tms`) are implicitly defined when the colorset is timed. They can not be explicitly declared.

ML Library

A number of declarations can be grouped in an ML structure. They can then be referred to by prefixing the name with `StructureName` followed by period. You can open the structure by writing `open StructureName;` and then referring to the items simply by means of their names.

When a text file contains a number of ML structures, the corresponding items are made accessible by writing

```
use "FileName"; open Struct1;.....; open StructN;
```

in the global or temporary declaration node.

The file `MLlib` contains a library of ML structures with additional constants, operations, and functions. These structures can be applied if you include the “use” command, described above, in a global or temporary declaration node. Please consult the file to see the items available.

Standard Constants, Operations, and Functions

The following constants, operations, and functions are part of the standard image called `cpn.ML`. Thus they may be used in inscriptions and code segments.

Since these are standard SML constructs, they are not defined in detail; the description simply identifies whether a particular construct is a constant, operator, or function and gives the SML datatype for the domain and range. In the case of constants it is the SML datatype of the constant. Minor differences for the two interpreters are specified as well.

CPN ML Reference

General Functions

Fun	o	(('a -> 'b) * ('d -> 'a)) -> ('d -> 'b)
Fun	use	string -> unit
Fun	usestring	(string list) -> unit
Fun	use_stream	instream -> unit (New Jersey)
Fun	open_string	string -> instream (New Jersey)

Boolean Functions

datatype **bool**

con	false	bool
con	true	bool
Fun	=	"a * "a -> bool
Fun	<>	"a * "a -> bool
Fun	makestring	bool -> string
Fun	not	bool -> bool

Integer Functions

datatype **int**

Fun	*	(int * int) -> int
Fun	+	(int * int) -> int
Fun	-	(int * int) -> int
Fun	<	(int * int) -> bool
Fun	<=	(int * int) -> bool
Fun	>	(int * int) -> bool
Fun	>=	(int * int) -> bool
Fun	abs	int -> int
Fun	div	(int * int) -> int
Fun	makestring	int -> string
Fun	max	(int * int) -> int
Fun	min	(int * int) -> int
Fun	mod	(int * int) -> int
Fun	~	int -> int

Real Functions

datatype **real**

Fun	*	(real * real) -> real
Fun	+	(real * real) -> real
Fun	-	(real * real) -> real
Fun	/	(real * real) -> real
Fun	<	(real * real) -> bool
Fun	<=	(real * real) -> bool
Fun	>	(real * real) -> bool
Fun	>=	(real * real) -> bool

Fun	abs	real -> real
Fun	arctan	real -> real
Fun	cos	real -> real
Fun	exp	real -> real
Fun	floor	real -> int
Fun	ln	real -> real
Fun	makestring	real -> string
Fun	min	(real * real) -> real
Fun	max	(real * real) -> real
Fun	real	int -> real
Fun	sin	real -> real
Fun	sqrt	real -> real
Fun	~	real -> real

String Functions

datatype **string**

Fun	<	(string * string) -> bool
Fun	<=	(string * string) -> bool
Fun	>	(string * string) -> bool
Fun	>=	(string * string) -> bool
Fun	^	(string * string) -> string
Fun	chr	int -> string
Fun	explode	string -> (string list)
Fun	implode	(string list) -> string
Fun	length	string -> int (Edinburgh)
Fun	ord	string -> int
Fun	ordof	(string * int) -> int
Fun	size	string -> int
Fun	substring	(string * int * int) -> string

List Functions

datatype 'a **list**

con	::	('a * ('a list)) -> ('a list)
con	nil	'a list
Fun	^^	(('a list) * ('a list)) -> ('a list)
Fun	app	('a -> 'b) -> (('a list) -> unit)
Fun	exists	(('a -> bool) * ('a list)) -> bool
Fun	fold	(('a * 'b) -> 'b) -> (('a list) -> ('b -> 'b))
Fun	hd	('a list) -> 'a
Fun	length	('a list) -> int
Fun	map	('a -> 'b) -> (('a list) -> ('b list))
Fun	nth	(('a list) * int) -> 'a
Fun	null	('a list) -> bool
Fun	rev	('a list) -> ('a list)
Fun	revapp	('a -> unit) -> (('a list) -> unit) (Edinburgh)
Fun	revapp	('a -> 'b) -> (('a list) -> unit) (New Jersey)
Fun	revfold	(('a * 'b) -> 'b) -> (('a list) -> ('b -> 'b))

CPN ML Reference

Fun **tl** ('a list) -> ('a list)

CPN ML uses ^^ as the operator symbol for list concatenation (instead of the standard @). If you are not using the time clause, you may reinstall the Standard name by typing "val @=^^;"

Reference Functions

datatype '**_a ref**

con **ref** '_a -> ('_a ref)
Fun **!** ('a ref) -> 'a
Fun **:=** (('a ref) * 'a) -> unit
Fun **dec** (int ref) -> unit
Fun **inc** (int ref) -> unit

I/O Functions

datatype **instream**
datatype **outstream**

exception **io_failure** string

Fun **close_in** instream -> unit
Fun **close_out** outstream -> unit
Fun **end_of_stream** instream -> bool
Fun **file_exists** string -> bool
Fun **input** (instream * int) -> string
Fun **input_line** instream -> string
Fun **lookahead** instream -> string
Fun **open_in** string -> instream
Fun **open_out** string -> outstream
Fun **open_append** string -> outstream
Fun **output** (outstream * string) -> unit
Fun **print** 'a -> 'a (* Writes to std_out *)
Fun **std_in** instream
Fun **std_out** outstream
Fun **log** outstream

Never call `close_in` with `std_in`. It will cause the system to malfunction.

Real Time Functions

The following function makes it possible to test with respect to the time shown by the operating system clock. The function returns the number of seconds elapsed since 00.00 GMT January 1, 1970.

Fun **tod** unit -> int

