# Upgrading from Mosel 5 to Mosel 6

October 2022

The new language features introduced by Mosel 6 are largely incremental to the existing functionality and language syntax. However, a few quite specific points relating to the use of undeclared list or set structures may require updates to existing Mosel source code.

## Types of undeclared lists

Mosel 6 allows the user to define lists that contain elements of different types, any such mixed type list now results in a `list of any`. With previous versions of Mosel a combination of the numerical types `real` and `integer` was permitted, resulting in the type `list of real`.

1. **Constant lists combining types `real` and `integer`:** In order to obtain the same type `list of real` with Mosel 6, all integer values need to be cast explicitly to `real` when defining constant lists of numerical values (`LR`) or be specified as real numbers (`LR2`):

```
declarations
  LC=[1,2.5]                ! Mosel 5: list of real; Mosel 6: list of any
  LR=[real(1),2.5]          ! Mosel 5+6: list of real
  LR2=[1.0,2.5]             ! Mosel 5+6: list of real
end-declarations
```

   As a consequence of the change of type for list `LC` in the code snippet above, Mosel 5 code accessing elements of this list, such as

```
declarations
  myval: real
end-declarations
myval:= LC(2)
```

   will result in an error message "Incompatible types in assignment" with Mosel 6.

2. **Non-constant lists combining types `real` and `integer`:** For non-constant lists that were previously undeclared and hence relied on the automatic type deduction it will be sufficient to add a suitable declaration:

```
declarations
  LD: list of real
end-declarations
LA:=[1,2.5]                 ! Mosel 5: list of real; Mosel 6: list of any
LD:=[1,2.5]                 ! Mosel 5+6: list of real
```

# Unnamed array index sets

Mosel 6 makes it possible to work with unnamed index sets in places where this was previously not possible, including in the definition of arrays that are used as fields in record structures. As a consequence of the revised set handling, a few cases where unnamed sets were already accepted in previous releases now behave in a more systematic way that is different from previous behaviour: unless two arrays specifically are declared to take the same (named) index, their sets are now always treated as being different, including for single line declarations of multiple arrays. As a result, code such as the following that works with previous versions of Mosel:

```
public declarations
  A,B: array(set of string) of integer
  data= `
data:[('a') [1 2] ('b') [3 4]]
 `
end-declarations

initialisations from 'mmsystem.text:data'
  [A,B] as 'data'
end-initialisations
```

will fail with the error message "Incompatible types for 'initialisations'". The correction consists in adding a name for the set in the array declaration in order to make sure that both arrays have indeed the same index set:

```
public declarations
  A,B: array(S: set of string) of integer
end-declarations
```

# Upgrading from Mosel 4 to Mosel 5

The new language features introduced by Mosel 5 are largely incremental to the existing functionality and language syntax. However, a few points may require updates to existing Mosel source code.

## *Dynamic package loading*

Packages are now handled like native modules: they are loaded dynamically at runtime when the model is loaded into memory.

### In models

In order to maintain the same behaviour with Mosel 5 as with Mosel 4, employ `imports` in place of `uses` for all packages that are not part of the Xpress distribution

- Make sure to always start with the `imports` statements, and then the `uses` statements (this is important if the same package is loaded several times, *e.g.* directly via a `uses` and indirectly through a dependency via `imports`).
- If you do decide to work with dynamic packages these must be made available at run time (in particular, this also concerns the BIM files in Insight apps).

### In packages

Specify the complete list of `uses` for all descendants that are used directly by this package.
Example: *pkga* uses *pkgb*, *pkgb* uses *pkgc* and *pkgd*
If *pkga* uses directly some functionality from *pkgc* it now needs to load this package explicitly, but there is no need for *pkga* to load *pkgd* if it does not use it.

- Do not use `imports` in packages.

### *mmjobs*

If a model or package loads a package via the *mmjobs* routine `load` (*e.g.* to inspect its annotations) you should now specify the option `'l'` for *lazy loading* to load and inspect only the package itself without any dependencies.

### Paths

Previously it was sufficient to specify BIM locations (*e.g.* via the BIM prefix `–bx`) for compilation, now these files are also required for loading a model (that is, at runtime) if the packages are loaded dynamically:

- Use the new environment variable `MOSEL_BIM` or set the control parameter `'bimprefix'` within a model:

```
setparam("bimprefix", "./")     ! Package BIMs are in the model's working directory
```

## *Array handling*

Mosel 5 has two categories of arrays: sparse arrays and dense arrays. *Sparse arrays* include explicitly `dynamic` arrays that keep the same behaviour as in previous versions and the new array type

`hashmap`. *Dense arrays* are all arrays that are not specifically marked as a sparse array type, namely static arrays (same behaviour as in previous versions) and not fixed (previously: implicitly dynamic) arrays that are now handled more consistently, always resulting in the same representation independent of index set finalization.

- Arrays for which index sets are not known at their declaration and that are not marked as `dynamic` or `hashmap` are now always represented as *dense* arrays independently of whether the index sets have been finalized (whereas previously they were represented as *sparse* arrays if the index sets were not finalized, and otherwise as *dense* arrays).
  ⇒ It is recommended to check your Mosel code for arrays that are not explicitly marked as `dynamic` and for which index sets are not known at the time of their creation since the default behaviour is changing in certain cases as shown below.

  - Mosel 3-4:

```
declarations
 RD: range
 D: dynamic array(RD) of real
 A2,A3: array(RD) of real
end-declarations

! Explicitly dynamic array
D(1):=10; D(-2):=-20; D(3):=150             ! Mosel 4 output:
writeln("D: ", D, "  RD: ", RD)             ! D: [(-2,-20),(1,10),(3,150)]  RD: -2..3
writeln("Size: ", D.size)                   ! Size: 3

! Array with dynamic index set
forall(i in RD | exists(D(i)) ) A2(i):=D(i)
writeln("A2: ", A2, " size=", A2.size)      ! A2: [(-2,-20),(1,10),(3,150)] size=3

! Finalize index set before array access
finalize(RD)
forall(i in RD | exists(D(i)) ) A3(i):=D(i)
writeln("A3: ", A3, " size=", A3.size)      ! A3: [-20,0,0,10,0,150] size=6
```

  - Mosel 5:

```
declarations
 RD: range
 D: dynamic array(RD) of real
 A2,A3: array(RD) of real
end-declarations

! Explicitly dynamic array
D(1):=10; D(-2):=-20; D(3):=150             ! Mosel 5 output:
writeln("D: ", D, "  RD: ", RD)             ! D: [(-2,-20),(1,10),(3,150)]  RD: -2..3
writeln("Size: ", D.size)                   ! Size: 3

! Array with dynamic index set
forall(i in RD | exists(D(i)) ) A2(i):=D(i)
writeln("A2: ", A2, " size=", A2.size)      ! A2: [-20,0,0,10,0,150] size=6

! Finalize index set before array access
finalize(RD)
forall(i in RD | exists(D(i)) ) A3(i):=D(i)
writeln("A3: ", A3, " size=", A3.size)      ! A3: [-20,0,0,10,0,150] size=6
```

- All entries corresponding to defined index values are created at once

  - previously: with dynamic index sets entries only for assigned values

```
declarations
 A: array(S: set of string) of real
end-declarations

S:={'a','b','c'}
A('c'):=1.5
writeln("A: ", A, " size=", A.size)    ! Output:
                                       ! Mosel 4:  A: [( `c',1.5)] size=1
                                       ! Mosel 5:  A: [0,0,1.5]    size=3
```

- Arrays grow if index sets increase in size
  - previously: no change

```
declarations
  B1,B2: array(SB:set of integer) of real
end-declarations

B1(1):=1.5; B1(4):=4.5
writeln("B1 size=", B1.size, "  B2 size=", B2.size)
                          ! Mosel 4:  B1 size=2  B2 size=0
                          ! Mosel 5:  B1 size=2  B2 size=2
writeln("B1:", B1, " B2:",B2, " SB:", SB)
                          ! Mosel 4:  B1:[(1,1.5),(4,4.5)]  B2:[]      SB:{1,4}
                          ! Mosel 5:  B1:[1.5,4.5]          B2:[0,0]   SB:{1,4}
! Growing index set
SB+={7,10}
writeln("B1 size=", B1.size, "  B2 size=", B2.size)
                          ! Mosel 4:  B1 size=2  B2 size=0
                          ! Mosel 5:  B1 size=4  B2 size=4
writeln("B1:", B1, " B2:",B2, " SB:", SB)
                          ! Mosel 4:  B1:[(1,1.5),(4,4.5)]  B2:[]          SB:{1,4,7,10}
                          ! Mosel 5:  B1:[1.5,4.5,0,0]      B2:[0,0,0,0]   SB:{1,4,7,10}
```

- There is no longer any need for using `create` on arrays of `mpvar` that are not explicitly `dynamic`

```
 public declarations
   x: array(I:range) of mpvar
   y: array(string) of mpvar
 end-declarations

 Ctr1:=sum(i in [1, 2, 5, 6]) i*x(i) <= 20
 Ctr2:=sum(i in ['a','b','c'], ct as counter) (ct+1)*y(i) <= 10

 exportprob
         ! Mosel 4:  empty problem (variables have not been created)
         ! Mosel 5:  _R1: 2 y(a) + 3 y(b) + 4 y(c) <= 10
         !           _R2: x(1) + 2 x(2) + 5 x(5) + 6 x(6) <= 20

 writeln("x size=", x.size, "  y size=", y.size)
         ! Mosel 4:  x size=0  y size=0
         ! Mosel 5:  x size=6  y size=3
```

- Use `finalize` on index sets in order to prevent the creation of additional (unwanted) entries

```
options noautofinal

public declarations
 COST: array(PERIODS:range) of real
 x: array(PERIODS) of mpvar
 Ctr: array(PERIODS) of linctr
end-declarations

initializations from "cost.dat"
 COST
```

```
end-initializations            !  COST: [(1) 3 (2) 6 (3) 9 (4) 6 (5) 2 (6) 2 ]

forall(t in PERIODS) create(x(t))    ! Required by Mosel 4, not Mosel 5

forall(t in PERIODS) Ctr(t):= x(t) >= x(t+1)
  ! This should really be:    x(t) >= if(t+1 in PERIODS, x(t+1), 0)

                                     ! Output Mosel 4:         Mosel 5:
exportprob("")                       ! Ctr(6): x(6) >= 0      Ctr(6): x(6) - x(7) >= 0
writeln("Periods: ", PERIODS)        ! Periods: 1..6          Periods: 1..7
```

- Use sparse (`dynamic` or `hashmap`) arrays when performing tests with `exists`, and also to prevent the creation of undesired entries (*e.g.* completion of ranges)

```
declarations
  A: array(R: range,set of integer) of integer
  DA: dynamic array(R,set of integer) of integer
end-declarations

writeln("A size=", A.size)                        ! Mosel 4: size=0   Mosel 5: size=0
A(10,10):=10; A(5,5):=5
writeln("A size=", A.size)                        ! Mosel 4: size=2   Mosel 5: size=12

writeln("DA size=", DA.size)                      ! Mosel 4: size=0   Mosel 5: size=0
DA(10,10):=10; DA(5,5):=5
writeln("DA size=", DA.size)                      ! Mosel 4: size=2   Mosel 5: size=2

writeln(A, " is dynamic:", isdynamic(A))
                             ! Mosel 4: [(5,5,5),(10,10,10)] is dynamic:false
                             ! Mosel 5: [0,5,0,0,0,0,0,0,0,0,10,0] is dynamic:false
writeln("A(7,10) exists:", exists(A(7,10)))       ! Mosel 4: false    Mosel 5: true
writeln("DA(7,10) exists:", exists(DA(7,10)))     ! Mosel 4: false    Mosel 5: false
```

- Use sparse arrays if you wish to delete entries with `delcell` or (new in Mosel 5) `reset`
  - for non-dynamic arrays these now result in resetting array contents to default values

```
declarations
  A,B: array(S:set of integer) of integer
  DA: dynamic array(S) of integer
end-declarations

writeln("A size=", A.size)                  ! Mosel 4: size=0   Mosel 5: size=0
S:={1,3}; A(4):=4.5; A(8):=8.5
writeln("A size=", A.size, "  A=", A)       ! Mosel 4: size=2  A=[(4,4.5),(8,8.5)]
                                            ! Mosel 5: size=4  A=[0,0,4.5,8.5]
delcell(A)    ! Same as:    reset(A)
writeln("A size=", A.size, "  A=", A)       ! Mosel 4: size=0  A=[]
                                            ! Mosel 5: size=4  A=[0,0,0,0]
finalize(S)
B(4):=4.5; B(8):=8.5
writeln("B size=", B.size, "  B=", B)       ! Mosel 4: size=4  B=[0,0,4.5,8.5]
                                            ! Mosel 5: size=4  B=[0,0,4.5,8.5]
delcell(B)    ! Same as:    reset(B)
writeln("B size=", B.size, "  B=", B)       ! Mosel 4: size=4  B=[0,0,4.5,8.5]
                                            ! Mosel 5: size=4  B=[0,0,0,0]

DA(4):=4.5; DA(8):=8.5
writeln("DA size=", DA.size, "  DA=", DA)   ! Mosel 4+5: size=2  DA=[(4,4.5),(8,8.5)]
delcell(DA(4))
writeln("DA size=", DA.size, "  DA=", DA)   ! Mosel 4+5: size=1  DA=[(8,8.5)]
delcell(DA)    ! Same as:    reset(DA)
writeln("DA size=", DA.size, "  DA=", DA)   ! Mosel 4+5: size=0  DA=[]
```

# Stricter syntax checks

Stricter syntax checks may reveal programming errors in certain cases.

## Requirement for the `public` qualifier

The requirement for the `public` qualifier is now made strict.

- The Mosel postprocessing API (C, C#, Java *etc.*) function `XPRMfindident` no longer works for entities that are not explicitly declared as `public`.
- When a user (record) type definition is public, all *types* used for fields in this record definition must be public (even if the fields themselves possibly are not public).

## Duplicate use of loop indices

The duplicate use of loop indices is no longer possible. For example, in place of

```
forall(f in union(f in Lsf) {f})
```

you now need to use

```
forall(f in union(ff in Lsf) {ff})
```

## Package names

Package names (that is, the name of the package BIM file) are identifiers. Whilst these names were previously not actively used, employing the same name for a package and for an entity, type, or subroutine is no longer possible.

# Compilation options

The Mosel compiler now uses by default the strip (-s) compilation and the previous default mode that was keeping private entities has been removed.

There will be no change in behaviour if you are using the -s compilation option for deployment/generation of production versions of your Mosel models following the standard recommendations for application deployment. However, if you have been working with the default compilation settings this new version might reveal cases where the explicit `public` qualifier is missing for entities that need to be visible to external programs (this includes Mosel subroutines employed as callbacks for Xpress Solvers and also any model entities that are accessed from Xpress Insight). Typical error messages in this case may look as follows:

```
XPRS: wrong procedure type for callback xxx
Kalis: Invalid function name passed to xxx
```

# Notes for Insight users

Xpress Insight 4.11.2 or later is compatible with Mosel 5.

App models compiled with Mosel 4.2 or more recent can be run with Mosel 5. However, although they have been compiled with an older release, they will be executing on the new architecture and their behavior may change due to the refactoring of the array handling (see Section *Array handling*).

If you re-compile the App model with Mosel 5, be aware of the following:

- All declarations of Insight entities need to be prefixed with the `public` qualifier (a quick workaround is to compile with the -g debug compilation option which preserves all global entities).
- Insight 4.x does not support dynamic packages. Use the `imports` keyword instead of `uses` to statically link in packages.
- Insight 4.x does not support the new namespaces feature.
- Mosel 5 introduces some changes to the handling of arrays (see Section *'Array handling'*) which may impact your model. The symptom is that a previously (implictly) dynamic array is now dense and this is best diagnosed by asserting whether the size of arrays and/or their dependencies such as problem matrix dimensions are changed.

**Note for Workbench users:** Workbench now compiles BIM files into an output folder in the project, by default `out`, and copies the main BIM file into the root of the archive only on publish or download app. Therefore any legacy BIM file in the root of a project is obsolete and should be deleted to avoid confusion.

# Upgrading from Mosel 3 to Mosel 4

## *Character encoding*

Mosel is now working in UTF-8. This concerns

- the internal representation of text
- all external APIs (*i.e.* all Mosel libraries)
- the communication with the system via Unicode (Windows) or system encoding (Posix)

All streams and text files default to UTF-8. There is no impact on applications that only use pure ASCII (first 127 characters), but *text data files and source code* using other encodings might require conversions or tagging.

Please note that the remarks on character encoding in this document only refer to text file formats, no changes are required for other file types including spreadsheets or databases.

### Character encoding — compatibility

If your model is no longer producing the expected output or not displaying correctly within IVE:

1. Use the command 'xprnls info' to identify which encoding is used on your system
   *Example:* output produced for western European Windows / 'latin' encoding (UK English):

```
C:\Test>xprnls info
Language: en
Default encodings:
 System:     CP1252
 Console:    CP437
 File names: CP1252
 Wide chars: UTF-16LE
```

2. Alternatively, use Xpress-IVE to display the system encoding from its menu entry *Help* ≫ *About Xpress-IVE*.
3. Specify the encoding (in model source and text data files in Mosel format) with the annotation `!@encoding`
4. Convert text/string input and output via the `enc:` prefix to file names and streams or by using the conversion routines of the XPRNLS library or command tool (see paragraph 'Character encoding — conversion' below)

### Character encoding — Example

- Previous behaviour (Mosel 3 and older):
  For the following Mosel model code

```
model "test characters"
  writeln_("Character test:  à é ß ö Ç £ ¥ ¿ ")
end-model
```

  on a system with western European configuration the correct output is displayed within IVE but not at the console:

```
C:\Test>mosel testchars
Character test:  α θ ■ ÷ ‖ ú Ñ ┐
```

- New behaviour with Mosel 4 (the placeholder '.' is shown instead of characters that cannot be properly represented):

```
C:\Test>mosel testchars
Character test:  . . . . . . . .
```

- Specifying the correct encoding at the beginning of the model source:

```
!@encoding CP1252
model "test characters"
  ...
```

results in this display:

```
C:\Test>mosel testchars
Character test:  à é ß ö Ç £ ¥ ¿
```

## Character encoding — conversion

- Text format data files (other than the Mosel `initializations` format for which the `!@encoding` marker can be used) such as CSV files or files accessed via `fopen` that do not use UTF-8 encoding need to be converted with the `'enc:'` prefix when accessing them from within a Mosel model.
  Example:

```
! Encoding names are operating system dependent, eg CP1252, ISO88591
fopen("enc:GB18030,testdata.txt", F_INPUT)
```

It is usually preferrable to specify the encoding used by a data file as shown above, but Mosel also implements shorthands for encodings configured on the system running the model.

```
! Encoding aliases:
! raw, sys, wchar, fname, tty, ttyin, stdin, stdout, stderr
initializations to "enc:sys,mmsheet.csv:testoutput.csv"
  ...
end-initializations
```

Using the prefix `enc:sys` re-establishes the previous behaviour of Mosel, namely defaulting to the system encoding.

- On the API level, you can use the *XPRNLS library* to convert to/from UTF-8 encoding:
  - this library is platform independent and has no external dependency
  - it handles encoding conversions between UTF-8 and local encodings
  - it implements UTF-8/16/32(LE+BE), ISO-8859-1/15, ASCII, CP1252
  - other supported encodings depend on the operating system

```
// Open a file using the C function 'fopen' with a file name
// coming from Mosel
f = fopen(XNLSconvstrto(XNLS_ENC_SYS,filename,-1,NULL),"r");
```

- Alternatively, you can use the *XPRNLS command tool* for converting the encoding of text files:

```
xprnls conv -f CP1252 -t UTF16 -o outfile.txt myfile.txt
```

# Upgrading from Mosel 2 to Mosel 3

This section describes the differences in language syntax, behaviour and the API between Mosel 3 and Mosel 2.X versions.

## *Language*

- 'count', 'counter' and 'with' are now keywords so these words and their all-uppercase form can no longer be used to name symbols.
- By default 'initialisations' blocks now finalize the sets they initialize and also the index sets of initialized non-dynamic arrays (*i.e.*, arrays not explicitly declared 'dynamic'). This may cause problems if the model modifies sets initialized via such a block or expects arrays of this kind always to be created dynamic. This automatic finalization functionality may be disabled by adding the following option to the model:

```
options noautofinal
```

## *Libraries*

- Deprecated routines 'XPRMgetdsoinfo' and 'XPRMgettypename' have been removed: 'XPRMgetdsoprop' and 'XPRMgettypeprop' must be used instead.
- Functions 'XPRMgetvarnum' and 'XPRMgetctrnum' now require a model reference.

## *Native Interface*

Mosel 3.0 accepts modules compiled for Mosel 2.4 but it no longer accepts modules compiled for Mosel 1.6. Upgrading requires the following changes:

- Deprecated routines 'getdsoinfo' and 'gettypename' have been removed: 'getdsoprop' and 'gettypeprop' must be used instead.
- Functions 'getvarnum' and 'getctrnum' now require a context.
- The type function 'fromstring' now expects an extra argument to return a pointer to the first character not used by the conversion.
- The function 'dsotypfromstr' now takes an extra argument to return a pointer to the first character not used by the conversion.

# Upgrading from Mosel 1 to Mosel 2

This section describes the differences in language syntax, behaviour and the API between Mosel 2 and previous versions.

## *Language*

- When a static array is implicitly created as dynamic and all its indexing sets are not empty, all possible entries are automatically created (in previous versions of Mosel this was the case only for '`mpvar`' arrays).
- Tuples have been replaced by lists; this may break code that exploited the fact that a tuple is represented by an array. For instance, assuming the procedure '`A`' is declared as follows:

```
procedure A(t:array(range)of integer)
```

  the following call works with Mosel 1.6 but cannot be compiled with Mosel 2.0:

```
A([1,2,3])
```

- The initialization operator '`::`' must now be used instead of the assignment operator '`:=`' for initializing arrays. For instance:

```
T:=[1,2,3,4]
T(2):=[5,6,7]   ! assuming T: array(1..10) of integer
```

  is now written:

```
T::[1,2,3,4]
T::(2..4) [5,6,7]
```

  This operator now requires the list of associated index values when indices are not ranges. For instance:

```
T:=[1,2,3,4]   ! assuming T:array({'a','b','c','d'}) of integer
```

  is now written:

```
T::(['a','b','c','d']) [1,2,3,4]
```

- The compiler now sorts constant sets that are known at compile time. As a consequence the order of enumeration of a set's elements may differ compared with Mosel 1.6. For instance the following statements give different output with Mosel 1.6 and Mosel 2.0:

```
forall(i in {4,3,2,1}) writeln(i)
S:={'c','a','b'}; writeln(S)
```

  It is possible to achieve similar behaviour to that of Mosel 1.x by using lists and the initialization operator '`::`':

```
forall(i in [4,3,2,1]) writeln(i)
S::['c','a','b']; writeln(S)
```

  Note however that sets are not ordered in Mosel and programs must not rely on the enumeration order of set elements.

- Parameters are now always public: the use of the 'public' qualifier in the 'parameters' block is now a syntax error.
- The notation '{x..y}' is now interpreted as a set of ranges: constant ranges must be written 'x..y'.
- Direct assignment between constraints now preserves the constraint type. For instance:

```
C:=x+y>=2
D:=C
```

  implies a duplicate constraint in the problem.

- Constraints generated for function parameters or lists are no longer added to the problem automatically. Such constraints are included in the problem if used as a statement or reassigned. For instance, assuming the following call:

```
myproc(x+y>=5)  ! this constraint is not stored
...
procedure myproc(c:linctr)
c               ! this adds the constraint to the problem
c+=10           ! same effect (after addition of a constant)
```

- The matrix generator is now deterministic *i.e.* it generates the same column ordering for a given model independently of the environment.
- The constraint 'is_integer' no longer sets an upper bound of MAX_INT.

## Libraries

- functions gettypename, getmodinfo and getdsoinfo are deprecated; use gettypeprop, getmodprop and getdsoprop instead.
- functions setstdin, setstdout, setstderr are no longer included ( deprecated since Mosel 1.4 ). Use the function setdefstream instead.
- Windows: the IO driver sysfd now requires file handles instead of C file descriptors (except for files 0, 1 and 2 that remain the default input, output and error streams).

## Native Interface

Mosel 2.0 accepts modules compiled for Mosel 1.6 and it is possible to compile code written for Mosel 1.6 with the new version by defining the macro USE_NI_16 before including xprm_ni.h. Upgrading requires the following changes:

- All type functions (create, delete, tostring, fromstring) now have an extra argument (this is the internal type number as an integer)
- The type creation function now has the following signature:

```
void *create(XPRMcontext,void *,void *, int);
```

  The extra pointer is used for handling reference counts.

- Although it is not mandatory, it is better to implement reference counting for module types (DTYP_RFCNT).

- The new `copy` operator is used when compiling assignments of arrays/records of user-defined types. These operations are disabled by the compiler if the`copy` function is not provided for the associated type.
- The service function `reset` now receives the requested module version as an extra parameter.
- Mosel may now release any local object when leaving a subroutine, including `mpvars`. A module routine that needs to keep references to objects must use the functions `newref` / `delref` to prevent Mosel deleting an object still in use.
- Tuples have been replaced by lists: module functions that used to get their parameters as tuples (=1- dimension array) must be modified to accept lists instead (signature '`l`' for list of any type and '`L?`' for list of type '?' [*e.g.* '`Li`'=list of integer]).