

Upgrading from Mosel 4 to Mosel 5

June 2019

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 `'1'` 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
writeln("D: ", D, " RD: ", RD)
writeln("Size: ", D.size)

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

! Mosel 5 output:
! D: [(-2,-20), (1,10), (3,150)] RD: -2..3
! 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

Ctrl1:=sum(i in [1, 2, 5, 6]) i*x(i) <= 20
Ctrl2:=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 lincpr
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 (implicitly) 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 master 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.