Multiple models and parallel solving with Mosel

6.10

WHITEPAPER

FICO® Xpress Optimization

FICO

FICO® Xpress Mosel 6.10 (FICO® Xpress 9.7)
Last Revised: 29 July, 2025

How to Contact the Xpress Team

Sales and Maintenance

If you need information on other Xpress Optimization products, or you need to discuss maintenance contracts or other sales-related items, contact FICO by:

- Phone: +1 (408) 535-1500 or +44 207 940 8718
- Web: www.fico.com/optimization and use the available contact forms

Product Support

Customer Self Service Portal (online support): www.fico.com/en/product-support

Email: Support@fico.com (Please include *'Xpress'* in the subject line)

For the latest news and Xpress software and documentation updates, please visit the Xpress website at http://www.fico.com/xpress or subscribe to our mailing list.

Multiple models and parallel solving with Mosel

Y. Colombani and S. Heipcke

Xpress Optimization, FICO, 6280 Bishops Court, Birmingham Business Park, Birmingham B37 7YB, UK
http://www.fico.com/xpress

Release 6.10

29 July, 2025

Abstract

This paper describes several examples of sequential and parallel solving of multiple models with FICO® Xpress Mosel. Without being able to give an exhaustive list of possible configurations, the examples showcase different uses of the Mosel module *mmjobs*, such as concurrent execution of several instances of a model, the (sequential) execution of submodels from another model with the alternative of embedding subproblems directly into this model, and the implementation of decomposition algorithms (Dantzig-Wolfe and Benders decomposition).

From a more technical point of view, topics discussed in this paper include model management, synchronization of concurrent models, and the use of the shared memory I/O driver.

We further discuss the difference between multiple *models* and the more recent functionality of multiple *problems* within a single model. Where appropriate (sequential solving) alternative implementations using multiple problems are described.

When working with several Mosel models the user may choose to distribute them over several instances of Mosel. We explain the functionality for remote execution of Mosel models as provided by the module *mmjobs* and document the corresponding modifications to the examples.

Contents

1 Introduction

Release 1.6 of Mosel introduced the possibility to work with multiple models directly in the Mosel language. The new functionality, provided by the module *mmjobs*, includes facilities for *model management*, *synchronization of concurrent models* based on event queues, and a *shared memory I/O driver*. Release 3.2 of Mosel (see [**?**]) extended the capacities for handling multiple models to distributed computing using several *Mosel instances* (running locally or on remote nodes connected through a network). The following list gives an overview on the available functionality. For the complete documentation of this module the reader is referred to the section *mmjobs* of the 'Mosel Reference Manual'.

- **Mosel instance management:** connecting and disconnecting Mosel instances, access to remote files, handling of host aliases.
- **Model management:** compilation of source model files, loading of bim files, model execution and interruption, retrieval of model information (status, exit code, ID), redirection of I/O streams.
- **Synchronization mechanism:** sending and retrieving events, waiting for events or event classes, retrieval of event information (class, value, sender model).
- **Shared memory I/O driver:** shared memory version of the `mem` driver for exchanging data between concurrent models (write access by a single model, read access by several models simultaneously), usable wherever Mosel expects a (generalized) filename, in particular in `initializations` blocks.
- **Memory pipe I/O driver:** memory pipes for exchanging data between concurrent models (write access by several models, read access by a single model), usable wherever Mosel expects a (generalized) filename, in particular in `initializations` blocks.
- **Remote connection I/O drivers:** two drivers for creating remote Mosel instances.
- **Remote file access I/O drivers:** access to physical files or streams on remote Mosel instances, usable wherever Mosel expects a (generalized) filename, in particular in `initializations` blocks.

mmjobs introduces three new types, `Mosel`, `Model` and `Event`. The type `Mosel` is used to reference a Mosel instance. Before an instance can execute commands (like loading or running a model), it must be connected, that is, an additional operating system process running Mosel must be started.
The type `Model` is used to reference a Mosel model. Before using the reference to a model it has to be initialized by loading a bim file.
The type `Event` represents an event in the Mosel language. Events are characterized by a *class* and a *value* and may be exchanged between a model and its *parent* model. An event queue is attached to each model to collect all events sent to this model and is managed with a FIFO policy (First In – First Out).

The first section of this paper introduces the reader to the basic tasks that are typically performed when working with several models in Mosel, namely:

- Executing a submodel from another model: the compile – load – run – wait sequence
- Stopping the submodel execution
- Output from the submodel: redirection to a file, making the submodel silent

- Compilation to memory
- Passing runtime parameters
- Running several submodels
  - in sequence
  - in parallel
- Communication of data between different models: using the shared memory and memory pipe I/O drivers
- Working with remote Mosel instances

The remainder of the paper gives some more advanced examples of the use of *mmjobs* with a detailed explanation of their implementation. All examples are available for download from the Xpress website.

- Column generation: re-implementation of the column generation example from the Mosel User Guide with two separate models that are solved sequentially, passing data via shared memory.
- Parallel solving: several instances of the same model are run concurrently with different solution algorithm parameterizations. Improved solution values are sent for bound updates to all running models and the first model that finishes stops all others.
- Dantzig-Wolfe decomposition: an iterative sequence of concurrent solving of a set of subproblem instances, followed by solving of the updated main problem; data exchange via shared memory.
- Benders decomposition: solving iteratively a sequence of several different subproblems; data exchange via shared memory.
- Start solution heuristic: working with cloned models sharing data structures; several concurrent submodel instances use the same input data as the main problem and communicate back results via shared data structures.

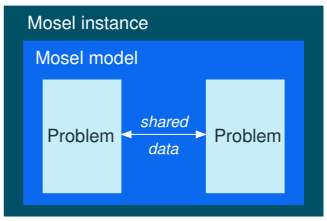
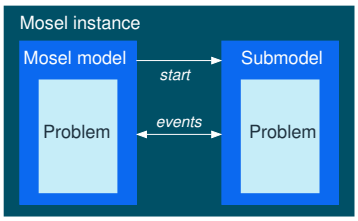
All the advanced examples are documented as standard (single-instance) models. However, multi-instance versions are provided with the set of example implementations.

1.1 Multi-problem vs. multi-model

At this place we would like to stress the difference between *multiple models* and *multiple problems* — Mosel releases prior to version 3.0 always associate a single problem with every model. This means, for instance, if a model contains several calls to a solver such as Xpress Optimizer, then the solver will work with a single problem representation, and only the solution to the last optimization run can be obtained from the solver at any time.

Release 3.0 of Mosel introduces the possibility of defining several *problems* within a single model. At any point a single problem is active. It is possible to switch back and forth between problems, *e.g.*, to retrieve solution information for a decision variable from different problems. To represent a 'problem', the Mosel language defines the type `mpproblem`, modules can provide their own problem types and users may equally create new problem types on the language level.

Here is a comparison of the main characteristics of multi-problem and multi-model implementations with Mosel.

***Multi-problem* (`mpproblem`)**	***Multi-model* (*mmjobs*)**
	
Single model file	Several model files
– problems share data	– communication of data (in memory)
– integrated; no direct access to (sub)problems by other models/applications	– stand-alone execution of submodels or use of submodels with other (parent) models/applications
Sequential access to problems only	Sequential or parallel execution of models

Problem solving approaches that involve parallel execution of (sub)models can only be implemented as multiple models, whereas sequential solving can be formulated with either one. For sequential algorithms (such as in Column generation and Benders decomposition) the developer may choose among the two design options.

2 Basic tasks

This section introduces the basic tasks that typically need to be performed when working with several models in Mosel using the functionality of the module *mmjobs*.

2.1 Executing a submodel

Assume we are given the following simple model `testsub.mos` (the *submodel*) that we wish to run from a second model (its *parent model*):

```
model "Test submodel"

 forall(i in 10..20) write(i^2, " ")
 writeln

end-model
```

The reader is certainly familiar with the standard `compile-load-run` sequence that is always required for the execution of a Mosel model independent of the place from where it is executed (be it the Mosel command line, a host application, or another Mosel model). In the case of a Mosel model executing a second model, we need to augment this standard sequence to `compile-load-run-wait`. The rationale behind this is that the submodel is started as a separate thread and we thus make sure that the submodel has terminated before the parent model ends (or continues its execution with results from the submodel, see Section 2.7 below).

The following model `runtestsub.mos` uses the `compile-load-run-wait` sequence in its basic form to execute the model `testsub.mos` printed above. The corresponding Mosel subroutines are defined by the module *mmjobs* that needs to be included with a `uses` statement. The submodel remains as is (that is, there is no need to include *mmjobs* if the submodel itself does not use any functionality of this module). The last statement of the controlling (parent) model, `dropnextevent`, may require some further explanation: the termination of the submodel is indicated by an event (of class `EVENT_END`) that is sent to the parent model. The `wait` statement pauses the execution of the parent model until it

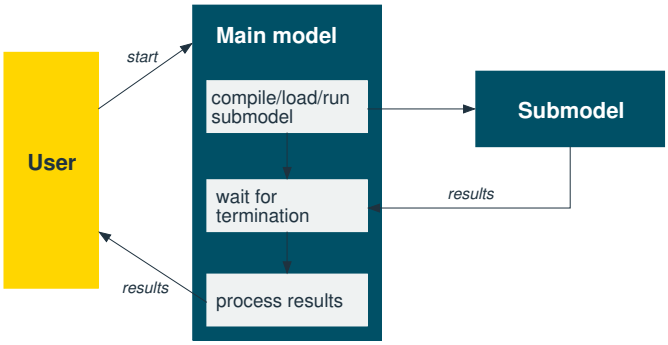


Figure 1: Executing a submodel

receives an event. Since in the present case the only event sent by the submodel is this termination message we simply remove the message from the event queue of the controlling model without checking its nature.

```
model "Run model testsub"
 uses "mmjobs"

 declarations
  modSub: Model
 end-declarations
                                    ! Compile the model file
 if compile("testsub.mos")<>0: exit(1)
 load(modSub, "testsub.bim")        ! Load the bim file
 run(modSub)                        ! Start model execution
 wait                               ! Wait for model termination
 dropnextevent                      ! Ignore termination event message

end-model
```

The two models are run by executing the controlling model in any standard way of executing Mosel models (from the Mosel command line, within Xpress Workbench, or from a host application). With the Mosel command line you may use, for instance, the following command:

```
mosel exec runtestsub.mos
```

As a result you should see the following output printed by the submodel (see Section 2.3 for a discussion of output handling):

```
100 121 144 169 196 225 256 289 324 361 400
```

2.1.1 Retrieving events

If we wish to obtain more precise information about the termination status of the submodel we could replace the statement `dropnextevent` by the following lines that retrieve the event sent by the submodel and print out its class (the termination event has the predefined class `EVENT_END`) and the value attached to it (here the default value 0). In addition, we display the exit code of the model (value sent by an `exit` statement terminating the model execution or the default value 0).

```
 declarations
  ev: Event
 end-declarations

 ev:=getnextevent
```

```
 writeln_("Event class: ", getclass(ev))
 writeln_("Event value: ", getvalue(ev))
 writeln_("Exit code  : ", getexitcode(modSub))
```

Events also provide information about the sending model, identified via its internal ID, and optionally the user ID and the group ID of the sending model (provided that the corresponding information has previously been defined for this model using `setuid` or `setgid` respectively):

```
 writeln_("Event sent by : ", getfromid(ev))
 writeln_("Model user ID : ", getfromuid(ev))
 writeln_("Model group ID: ", getfromgid(ev)
```

2.2 Stopping the submodel execution

If a submodel execution takes a long time it may be desirable to interrupt the submodel without stopping the controlling model itself. The following modified version of our parent model (file `runsubwait.mos`) shows how this can be achieved by adding a duration (in seconds) to the `wait` statement. If the submodel has not yet sent the termination event message after executing for one second it is stopped by the call to `stop` with the model reference.

```
model "Run model testsub"
 uses "mmjobs"

 declarations
  modSub: Model
 end-declarations
                                ! Compile the model file
 if compile("testsub.mos")<>0: exit(1)
 load(modSub, "testsub.bim")    ! Load the bim file
 run(modSub)                    ! Start model execution
 wait(1)                        ! Wait 1 second for an event

 if isqueueempty then           ! No event has been sent: model still runs
  writeln_("Stopping the submodel")
  stop(modSub)                  ! Stop the model
  wait                          ! Wait for model termination
 end-if
 dropnextevent                  ! Ignore termination event message

end-model
```

A more precise time measurement can be obtained by retrieving a ”model ready” *user event* from the submodel before we begin to `wait` for a given duration. With heavy operating system loads the actual submodel start may be delayed, and the event sent by the submodel tells the parent model the exact point of time when its processing is started. Class codes for user events can take any integer value greater than 1 (values 0 and 1 are reserved respectively for the `nullevent` and the predefined class `EVENT_END`).

```
model "Run model testsub"
 uses "mmjobs"

 declarations
  modSub: Model
  ev: Event
  SUBMODREADY = 2               ! User event class code
 end-declarations
                                ! Compile the model file
 if compile("testsubev.mos")<>0: exit(1)
 load(modSub, "testsubev.bim")  ! Load the bim file
 run(modSub)                    ! Start model execution
```

```
 wait                                ! Wait for an event
 if getclass(getnextevent) <> SUBMODREADY then
  writeln_("Problem with submodel run")
  exit(1)
 end-if

 wait(1)                             ! Let the submodel run for 1 second

 if isqueueempty then                ! No event has been sent: model still runs
  stop(modSub)                       ! Stop the model
  wait                               ! Wait for model termination
 end-if

 ev:=getnextevent                    ! An event is available: model finished
 writeln_("Event class: ", getclass(ev))
 writeln_("Event value: ", getvalue(ev))
 writeln_("Exit code  : ", getexitcode(modSub))

end-model
```

The modified submodel `testsubev.mos` now looks as follows. The user event class must be the same as in the parent model. In this example we are not interested in the value sent with the event and therefore simply leave it at 0.

```
model "Test submodel (Event)"
 uses "mmjobs"

 declarations
  SUBMODREADY = 2                    ! User event class code
 end-declarations

 send(SUBMODREADY, 0)                ! Send "submodel ready" event

 forall(i in 10..20) write(i^2, " ")
 writeln

end-model
```

2.3 Output from the submodel

By default, submodels use the same location for their output as their parent model. This implies that, when several (sub)models are run in parallel their output, for example on screen, is likely to mix up. The best way to handle this situation is to redirect the output from each model to a separate file.

Output may be redirected directly within the submodel with statements like the following added to the model before any output is printed:

```
 fopen("testout.txt", F_OUTPUT+F_APPEND) ! Output to file (in append mode)
 fopen("tee:testout.txt&", F_OUTPUT)     ! Output to file and on screen
 fopen("null:", F_OUTPUT)                ! Disable all output
```

where the first line redirects the output to the file `testout.txt`, the second statement maintains the output on screen while writing to the file at the same time, and the third line makes the model entirely silent. The output file is closed by adding the statement `fclose(F_OUTPUT)` after the printing statements in the model.

The same can be achieved from the parent model by adding output redirection before the `run` statement for the corresponding submodel ('modSub'), such as:

```
 setdefstream(modSub, F_OUTPUT, "testout.txt") ! Output to file
 setdefstream(modSub, F_OUTPUT, "tee:testout.txt&")
```

```
                                              ! Output to file and on screen
 setdefstream(modSub, F_OUTPUT, "null:")      ! Disable all output
```

The output redirection for a submodel may be terminated by resetting its output stream to the default output:

```
 setdefstream(modSub, F_OUTPUT, "")
```

2.4 Compilation to memory

The default compilation of a Mosel file *filename*`.mos` generates a binary model file *filename*`.bim`. To avoid the generation of physical BIM files for submodels we may compile the submodel to memory, as shown in the following example `runsubmem.mos`. Working in memory usually is more efficient than accessing physical files. Furthermore, this feature will also be helpful if you do not have write access at the place where the parent model is executed.

```
model "Run model testsub"
 uses "mmjobs", "mmsystem"

 declarations
  modSub: Model
 end-declarations
                                  ! Compile the model file
 if compile("", "testsub.mos", "shmem:testsubbim")<>0 then
  exit(1)
 end-if
 load(modSub, "shmem:testsubbim")  ! Load the bim file from memory
 fdelete("shmem:testsubbim")       ! ... and release the memory block
 run(modSub)                       ! Start model execution
 wait                              ! Wait for model termination
 dropnextevent                     ! Ignore termination event message

end-model
```

The full version of `compile` takes three arguments: the compilation flags (*e.g.*, use `"g"` for debugging), the model file name, and the output file name (here a label prefixed by the name of the shared memory driver). Having loaded the model we may free the memory used by the compiled model with a call to `fdelete` (this subroutine is provided by the module *mmsystem* that needs to be loaded in addition to *mmjobs*).

2.5 Runtime parameters

A convenient means of modifying data in a Mosel model when running the model (that is, without having to modify the model itself and recompile it) is to use *runtime parameters*. Such parameters are declared at the beginning of the model in a `parameters` block where every parameter is given a default value that will be applied if no other value is specified for this parameter at the model execution.

Consider the following model `rtparams.mos` that may receive parameters of four different types—integer, real, string, and Boolean—and prints out their values.

```
model "Runtime parameters"
 parameters
  PARAM1 = 0
  PARAM2 = 0.5
  PARAM3 = ''
  PARAM4 = false
 end-parameters
```

```
 writeln_(PARAM1, "  ", PARAM2, "  ", PARAM3, "  ", PARAM4)

end-model
```

The model `runrtparam.mos` executing this (sub)model may look as follows—all runtime parameters are given new values:

```
model "Run model rtparams"
 uses "mmjobs"

 declarations
  modPar: Model
 end-declarations
                                  ! Compile the model file
 if compile("rtparams.mos")<>0: exit(1)
 load(modPar, "rtparams.bim")     ! Load the bim file
                                  ! Start model execution
 run(modPar, "PARAM1=" + 2 + ",PARAM2=" + 3.4 +
             ",PARAM3='a string'" + ",PARAM4=" + true)
 wait                             ! Wait for model termination
 dropnextevent                    ! Ignore termination event message

end-model
```

An alternative formulation to the above makes use of the subroutine `setmodpar` to define the parameter values for the submodel (notice that we need to load the module *mmsystem* for the text handling functionality)—after starting the model run we delete the parameters definition; it would also be possible to remove individual parameter value settings by applying `resetmodpar` to the object `params`.

```
model "Run model rtparams 2"
 uses "mmjobs", "mmsystem"

 declarations
  modPar: Model
  params: text
 end-declarations
                                  ! Compile the model file
 if compile("rtparams.mos")<>0: exit(1)
 load(modPar, "rtparams.bim")     ! Load the bim file
 setmodpar(params, "PARAM1", 2)   ! Set values for model parameters
 setmodpar(params, "PARAM2", 3.4)
 setmodpar(params, "PARAM3", 'a string')
 setmodpar(params, "PARAM4", true)
 run(modPar, params)              ! Start model execution
 params:= ""                      ! Delete the parameters definition
 wait                             ! Wait for model termination
 dropnextevent                    ! Ignore termination event message

end-model
```

2.6 Running several submodels

Once we have seen how to run a parameterized model from another model it is only a small step to the execution of several different submodel instances from a Mosel model. The following sections deal with the three cases of sequential, fully parallel, and restricted (queued) parallel execution of submodels. For simplicity's sake, the submodels in our examples are all parameterized versions of a single model. It is of course equally possible to compile, load, and run different submodel files from a single parent model.

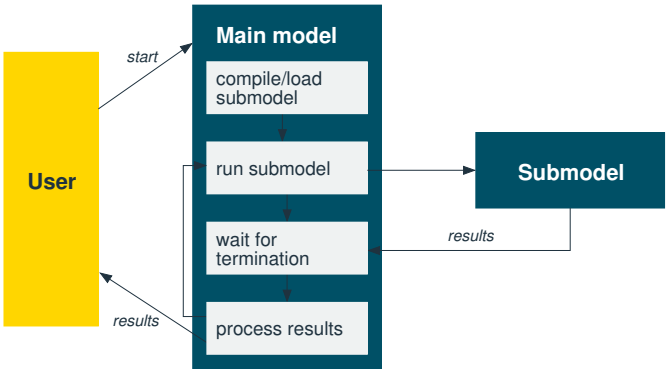


Figure 2: Sequential submodels

2.6.1 Sequential submodels

Running several instances of a submodel in sequence only requires small modifications to the parent model that we have used for a single model instance as can be seen from the following example (file `runrtparamseq.mos`)—to keep things simple, we now only reset a single parameter at every execution:

```
model "Run model rtparams in sequence"
 uses "mmjobs"

 declarations
  A = 1..10
  modPar: Model
 end-declarations
                                    ! Compile the model file
 if compile("rtparams.mos")<>0: exit(1)
 load(modPar, "rtparams.bim")       ! Load the bim file

 forall(i in A) do
  run(modPar, "PARAM1=" + i)        ! Start model execution
  wait                              ! Wait for model termination
  dropnextevent                     ! Ignore termination event message
 end-do

end-model
```

The submodel is compiled and loaded once and after starting the execution of a submodel instance we wait for its termination before the next instance is started.

2.6.2 Parallel submodels

The parallel execution of submodel (instances) requires slightly more modifications. We still have to compile the submodel only once, but it now needs to be loaded as many times as we want to run parallel instances. The `wait` statements are now moved to a separate loop since we first want to start all submodels and then wait for their termination.

```
model "Run model rtparams in parallel"
 uses "mmjobs"

 declarations
  A = 1..10
  modPar: array(A) of Model
 end-declarations
```

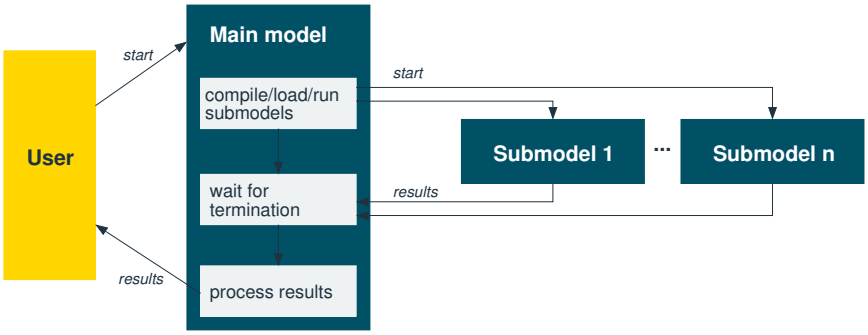


Figure 3: Parallel submodels

```
                                    ! Compile the model file
 if compile("rtparams.mos")<>0: exit(1)

 forall(i in A) do
  load(modPar(i), "rtparams.bim")   ! Load the bim file
  run(modPar(i), "PARAM1=" + i)     ! Start model execution
 end-do

 forall(i in A) do
  wait                              ! Wait for model termination
  dropnextevent                     ! Ignore termination event message
 end-do

end-model
```

The order in which the submodel output appears on screen is nondeterministic because the models are run in parallel. However, since the submodel execution is very quick, this may not become obvious: try adding the line `wait(1)` to the submodel `rtparams.mos` immediately before the `writeln` statement (you will also need to add the statement `uses "mmjobs"` at the beginning of the model) and compare the output of several runs of the main model. You are now likely to see different output sequences with every run.

Instead of repeatedly loading the submodel instances from the same BIM file we only really need to do so once and can then produce all further copies as *clones* of the first submodel—doing so will generally be slightly more efficient in terms of speed and memory usage.

```
model "Run model rtparams in parallel with cloned submodels"
 uses "mmjobs"

 declarations
  A = 1..10
  modPar: array(A) of Model
 end-declarations
                                    ! Compile the model file
 if compile("rtparams.mos")<>0: exit(1)

 forall(i in A) do
  if i=1 then
   load(modPar(i), "rtparams.bim") ! First submodel: Load the bim file
  else
   load(modPar(i), modPar(1))      ! Clone the already loaded bim
  end-if

  run(modPar(i), "PARAM1=" + i)     ! Start model execution
 end-do
```

```
 forall(i in A) do
  wait                              ! Wait for model termination
  dropnextevent                     ! Ignore termination event message
 end-do

end-model
```

2.6.3 Job queue for managing parallel submodels

The parallel execution of submodels in the previous section starts a large number of models at the same time. With computationally more expensive submodel runs this simple design might not be an appropriate choice: as a general rule, we recommend that the number of concurrent (sub)models should not exceed the number of processors available to avoid any negative impact on performance, and there might also be restrictions on the number of concurrent models imposed by your Xpress licence.

The following example therefore shows how to extend the parent model from the previous section as to limit the number of parallel submodel executions. The model instances to be run are implemented as a job queue, a list of instance indices that is accessed in first-in-first-out order. There are now two sets of indices, the *job indices* (set `A`) for the instances we wish to process and the *model indices* (set `RM`) corresponding to the concurrently executing Mosel models. We save the model index information as *user model ID* with each model and use the mapping `jobid` that relates the job indices to the model indices.

As before, we compile the submodel and load the required number of model instances into Mosel. We then take the first few entries from the job list and start the corresponding model runs. As soon as a model run terminates, a new instance is started through the procedure `start_next_job` (see below).

The model takes a parameter `NUMPAR` that lets you change the limit on the number of submodels at run time.

```
model "Run model rtparams with job queue"
 uses "mmjobs"

 parameters
   NUMPAR=2                         ! Number of parallel model executions
 end-parameters                     ! (preferrably <= no. of processors)

 forward procedure start_next_job(submod:Model)

 declarations
   RM = 1..NUMPAR                   ! Model indices
   A = 1..10                        ! Job (instance) indices
   modPar: array(RM) of Model       ! Models
   jobid: array(set of integer) of integer  ! Job index for model IDs
   JobList: list of integer         ! List of jobs
   JobsRun: set of integer          ! Set of finished jobs
   JobSize: integer                 ! Number of jobs to be executed
   Msg: Event                       ! Messages sent by models
 end-declarations
                                    ! Compile the model file
 if compile("rtparams.mos")<>0: exit(1)

 forall(m in RM) do
   load(modPar(m), "rtparams.bim")  ! Load the bim file
   modPar(m).uid:= m                ! Store the model ID as UID
 end-do

 JobList:= sum(i in A) [i]          ! Define the list of jobs (instances)
 JobSize:=JobList.size              ! Store the number of jobs
 JobsRun:={}                        ! Set of terminated jobs is empty

!**** Start initial lot of model runs ****
 forall(m in RM)
```

```
   if JobList<>[] then
     start_next_job(modPar(m))
   end-if

!**** Run all remaining jobs ****
 while (JobsRun.size<JobSize) do
   wait                              ! Wait for model termination
   Msg:= getnextevent
   if getclass(Msg)=EVENT_END then   ! We are only interested in "end" events
     m:=getfromuid(Msg)              ! Retrieve the model UID
     JobsRun+={jobid(m)}             ! Keep track of job termination
     writeln_("End of job ", jobid(m), " (model ", m, ")")
     if JobList<>[] then             ! Start a new run if queue not empty
      start_next_job(modPar(m))
     end-if
   end-if
 end-do

end-model
```

In the `while` loop above we identify the model that has sent the termination message and add the corresponding job identifier to the set `JobsRun` of completed instances. If there are any remaining jobs in the list, the procedure `start_next_job` is called that takes the next job and starts it with the model that has just been released.

```
 procedure start_next_job(submod: Model)
   i:=getfirst(JobList)              ! Retrieve first job in the list
   cuthead(JobList,1)                ! Remove first entry from job list
   jobid(submod.uid):= i
   writeln_("Start job ", i, " (model ", submod.uid, ")")
   run(submod, "PARAM1=" + i + ",PARAM2=" + 0.1*i +
               ",PARAM3='string " + i + "'" + ",PARAM4=" + isodd(i))
 end-procedure
```

The job queue in our example is static, in the sense that the list of jobs to be processed is fixed right at the beginning. Using standard Mosel list handling functionality, it can quite easily be turned into a dynamic queue to which new jobs are added while others are already being processed.

2.7 Communication of data between models

Runtime parameters are a means of communicating single data values to a submodel but they are not suited, for instance, to pass data tables or sets to a submodel. Also, they cannot be employed to retrieve any information from a submodel or to exchange data between models during their execution. All these tasks are addressed by the two I/O drivers defined by the module *mmjobs*: the `shmem` driver and the `mempipe` driver. As was already stated earlier, the `shmem` driver is meant for one-to-many communication (one model writing, many reading) and the `mempipe` driver serves for many-to-one communication. In the case of one model writing and one model reading we may use either, where `shmem` is conceptionally probably the easier to use.

2.7.1 Using the shared memory driver

With the `shmem` shared memory driver we write and read data blocks from/to memory. The use of this driver is quite similar to the way we would work with physical files. We have already encountered an example of its use in Section 2.4: the filename is replaced by a label, prefixed by the name of the driver, such as `"mmjobs.shmem:aLabel"`. If the module *mmjobs* is loaded by the model (or another model held in memory at the same time, such as its parent model) we may use the short form `"shmem:aLabel"`. Another case where we would have to explicitly add the name of a module to a driver occurs when we need to distinguish between several `shmem` drivers defined by different modules.

The exchange of data between different models is carried out through `initializations` blocks. In general, the `shmem` driver will be combined with the `bin` driver to save data in binary format (an alternative, somewhat more restrictive binary format is generated by the `raw` driver).

Let us now take a look at a modified version `testsubshm.mos` of our initial test submodel (Section 2.1). This model reads in the index range from memory and writes back the resulting array to memory:

```
model "Test submodel"
 declarations
  A: range
  B: array(A) of real
 end-declarations

 initializations from "bin:shmem:indata"
  A
 end-initializations

 forall(i in A) B(i):= i^2

 initializations to "bin:shmem:resdata"
  B
 end-initializations

end-model
```

The model `runsubshm.mos` to run this submodel may look as follows:

```
model "Run model testsubshm"
 uses "mmjobs"

 declarations
  modSub: Model
  A = 30..40
  B: array(A) of real
 end-declarations
                                  ! Compile the model file
 if compile("testsubshm.mos")<>0: exit(1)
 load(modSub, "testsubshm.bim")    ! Load the bim file

 initializations to "bin:shmem:indata"
  A
 end-initializations

 run(modSub)                      ! Start model execution
 wait                             ! Wait for model termination
 dropnextevent                    ! Ignore termination event message

 initializations from "bin:shmem:resdata"
  B
 end-initializations

 writeln(B)

end-model
```

Before the submodel run is started the index range is written to memory and after its end we retrieve the result array to print it out from the parent model.

If memory blocks are no longer used it is recommended to free up these blocks by calling `fdelete` (subroutine provided by module *mmsystem*), especially if the data blocks are large, since the data blocks survive the termination of the model that has created them, even if the model is unloaded explicitly, until the module *mmjobs* is unloaded (explicitly or by the termination of the Mosel session). At the end of our parent model we might thus add the lines

```
 fdelete("shmem:A")
 fdelete("shmem:B")
```

2.7.2 Using the memory pipe driver

The memory pipe I/O driver `mempipe` works in the opposite way to what we have seen for the shared memory driver: a pipe first needs to be opened before it can be written to. That means we need to call `initializations from` before `initializations to`. The submodel (file `testsubpip.mos`) now looks as follows:

```
model "Test submodel"
 declarations
  A: range
  B: array(A) of real
 end-declarations

 initializations from "mempipe:indata"
  A
 end-initializations

 forall(i in A) B(i):= i^2

 initializations to "mempipe:resdata"
  B
 end-initializations

end-model
```

This is indeed not very different from the previous submodel. However, there are more changes to the parent model (file `runsubpip.mos`): the input data is now written by this model *after* the submodel run has started. The parent model then opens a new pipe to read the result data *before* the submodel terminates its execution.

```
model "Run model testsubpip"
 uses "mmjobs"

 declarations
  modSub: Model
  A = 30..40
  B: array(A) of real
 end-declarations
                                ! Compile the model file
 if compile("testsubpip.mos")<>0: exit(1)
 load(modSub, "testsubpip.bim")    ! Load the bim file

 run(modSub)                    ! Start model execution

 initializations to "mempipe:indata"
  A
 end-initializations

 initializations from "mempipe:resdata"
  B
 end-initializations

 wait                           ! Wait for model termination
 dropnextevent                  ! Ignore termination event message

 writeln(B)

end-model
```

Once a file has opened a pipe for reading it remains blocked in this state until it has received the

requested data through this pipe. The program control flow is therefore the following in the present case:

1. The controlling (parent) model starts the submodel.
2. The submodel opens the input data pipe and waits for its parent to write to it.
3. Once the input data has been communicated the submodel continues its execution while the parent model opens the result data pipe and waits for the results.
4. When the result data pipe is open, the submodel writes to the result data pipe and then terminates.
5. The controlling model prints out the result.

2.7.3 Shared data structures for cloned models

For the special case of submodels that are a clone of the controlling model itself (that is, the `load` for the submodel instance duplicates the parent model in the same Mosel instance instead of reading in another BIM file) it is possible to declare `shared` data structures that are immediately accessible for all clones without any copying via `initializations from/to`. Note that only arrays, lists and sets of basic types (integer, real, boolean, string) can be shared between cloned models and the structure of the data cannot be changed while they are shared.

- It is possible to assign new values to existing entries of an array, but it is not possible to add or remove entries from an array.
- Sets and lists cannot be modified while being shared.

The parameter `sharingstatus` indicates the sharing status of a model which is one of: -1 (the model does not share any data), 0 (the model shares data but no submodel is using it), 1 (shared data is in use), 2 (the model is a submodel using shared data).

If we wish to work with this functionality for the small example from the previous section, we need to implement the parent and submodel in a single file `runsubclone.mos` as shown below.

```
model "Sharing data between clones"
 uses "mmjobs"

 declarations
  modSub: Model
  A = 30..40                          ! Constant sets are shared
  B: shared array(A) of real          ! Shared array structure
 end-declarations

 if getparam("sharingstatus")<=0 then
  ! **** In main model ****
  writeln_("In main: ", B)            ! *** Output:  In main: [0,...0]
  load(modSub)                        ! Clone the current model
  run(modSub)                         ! Run the submodel...
  waitforend(modSub)                  ! ...and wait for its end
  writeln_("After sub: ", B)          ! *** Output:  After sub: [900,...1600]

 else
  ! **** In submodel ****
  writeln_("In sub: ", B)             ! *** Output:  In sub: [0,...0]
  forall(i in A) B(i):= i^2           ! Modify the shared data

 end-if

end-model
```

2.8 Working with remote Mosel instances

The remote execution of submodels only requires few additions to models running submodels on a single Mosel instance. The examples in this section show how to extend a selection of the models we have seen so far with distributed computing functionality. Before you access Mosel on a remote host, or an additional local instance of Mosel, you need to start up the *Mosel server* (on the node you wish to use) with the command

```
xprmsrv
```

This command may take some options, such as the verbosity level, and the TCP port to be used by connections. The options are documented in the section on *mmjobs* in the Mosel Language Reference Manual, and you can also display a list with

```
xprmsrv -h
```

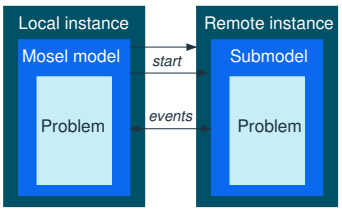


Figure 4: Remote submodel execution

2.8.1 Executing a submodel remotely

To execute a model remotely, we first need to create a new Mosel instance on the remote node. In this `connect` statement, the string that identifies the remote machine may be its local name, an IP address, or the empty string for the current node. In the example `runrtdistr.mos` below, we compile the submodel locally and after connecting, load the BIM file into the remote instance (notice the use of the `rmt` I/O driver to specify that the BIM is located on the root node). The `load` subroutine now takes an additional first argument indicating the Mosel instance we want to use. All else, including the `run` statement and the handling of events remain the same as in the single instance version of this model from Section 2.5.

```
model "Run model rtparams remotely"
 uses "mmjobs"

 declarations
  modPar: Model
  mosInst: Mosel
 end-declarations
                                  ! Compile the model file
 if compile("rtparams.mos")<>0: exit(1)

 NODENAME:= ""                    ! "" for current node, or name, or IP address
                                  ! Open connection to a remote node
 if connect(mosInst, NODENAME)<>0: exit(2)
                                  ! Load the bim file
 load(mosInst, modPar, "rmt:rtparams.bim")
                                  ! Start model execution + parameter settings
 run(modPar, "PARAM1=" + 2 + ",PARAM2=" + 3.4 +
             ",PARAM3='a string'" + ",PARAM4=" + true)
 wait                             ! Wait for model termination
 dropnextevent                    ! Ignore termination event message
```

```
end-model
```

Alternatively to the above, we might have compiled the submodel remotely after having established the connection, for instance using this code (the `rmt` prefix now appears in the compilation since we assume that the submodel is located at the same place as its parent model).

```
if connect(mosInst, NODENAME)<>0: exit(2)
if compile(mosInst, "", "rmt:rtparams.mos", "rtparams.bim")<>0 then
 exit(1); end-if
load(mosInst, modPar, "rtparams.bim")
```

Compiling the submodel on the original (root) node is preferable if we wish to execute several model instances on different remote nodes, and it may also be necessary if write access is not permitted on the remote node.

By default, the `rmt` driver refers to the root node. It is also possible to prefix the filename by its node number enclosed in square brackets, the special node number `-1` denotes the immediate parent node, such as in

```
load(mosInst, modPar, "rmt:[-1]rtparams.bim")
```

Note: In this example, we assume that we know of a specific Mosel instance that we wish to connect to. The example model in Section 2.8.4 shows how to search for available *xprmsrv* servers on the local network.

2.8.2 Parallel submodels in distributed architecture

The following model `runrtpardistr.mos` is an extension of the example from Section 2.6.2. The 10 model instances to be run are now distributed over 5 Mosel instances, assigning 2 models per instance.

```
model "Run model rtparams in distributed architecture"
 uses "mmjobs", "mmsystem"

 declarations
  A = 1..10
  B = 1..5
  modPar: array(A) of Model
  moselInst: array(B) of Mosel
  NODENAMES: array(B) of string
 end-declarations

                                !!! Select the (remote) machines to be used:
    !!! Use names, IP addresses, or empty string
    !!! for the node running this model
 forall(i in B) NODENAMES(i):= ""

                                  ! Compile the model file locally on root
 if compile("rtparams3.mos")<>0: exit(1)

 instct:=0
 forall(i in A) do
  if isodd(i) then
   instct+=1                      ! Connect to a remote machine
   if connect(moselInst(instct), NODENAMES(instct))<>0: exit(2)
  end-if

  writeln_("Current node: ", getsysinfo(SYS_NODE),
           " submodel node: ", getsysinfo(moselInst(instct), SYS_NODE))

                                  ! Load the bim file (located at root node)
```

```
  load(moselInst(instct), modPar(i), "rmt:rtparams3.bim")
                                  ! Start remote model execution
  run(modPar(i), "PARAM1=" + i + ",PARAM2=" + 0.1*i +
                 ",PARAM3='string " + i + "'" + ",PARAM4=" + isodd(i))
 end-do

 forall(i in A) do
  wait                            ! Wait for model termination
  dropnextevent                   ! Ignore termination event message
 end-do

end-model
```

We work here with a slightly modified version `rtparams3.mos` of the submodel that has some added reporting functionality, displaying node and model numbers and the system name of the node.

```
model "Runtime parameters"
 uses "mmjobs", "mmsystem"
 parameters
  PARAM1 = 0
  PARAM2 = 0.5
  PARAM3 = ''
  PARAM4 = false
 end-parameters

 writeln_("Node: ", getparam("NODENUMBER"), " ", getsysinfo(SYS_NODE),
          ". Parent: ", getparam("PARENTNUMBER"),
          ". Model number: ", getparam("JOBID"),
          ". Parameters: ",
          PARAM1, "  ", PARAM2, "  ", PARAM3, "  ", PARAM4)
end-model
```

2.8.3 Job queue for parallel execution in a distributed architecture

Let us now take a look at how we can extend the job queue example from Section 2.6.3 to the case of multiple Mosel instances. We shall work with a single queue (list) of jobs, from which we take the first available job as soon as another model has terminated and we start this new model on the corresponding Mosel instance. Each instance of Mosel processes at most `NUMPAR` submodels at any time.

```
model "Run model rtparams with job queue"
 uses "mmjobs", "mmsystem"

 parameters
   J=10                           ! Number of jobs to run
   NUMPAR=2                       ! Number of parallel model executions
 end-parameters                   ! (preferrably <= no. of processors)

 forward procedure start_next_job(submod: Model)

 declarations
   RM: range                      ! Model indices
   JOBS = 1..J                    ! Job (instance) indices
   modPar: array(RM) of Model     ! Models
   jobid: array(set of integer) of integer  ! Job index for model UIDs
   JobList: list of integer       ! List of jobs
   JobsRun: set of integer        ! Set of finished jobs
   JobSize: integer               ! Number of jobs to be executed
   Msg: Event                     ! Messages sent by models
   NodeList: list of string       !
   nodeInst: array(set of string) of Mosel  ! Mosel instances on remote nodes
   nct: integer
   modNode: array(set of integer) of string ! Node used for a model
   MaxMod: array(set of string) of integer
```

```
 end-declarations
                                    ! Compile the model file locally
 if compile("rtparams.mos")<>0: exit(1)

!**** Setting up remote Mosel instances ****
 sethostalias("localhost2","localhost")
 NodeList:= ["localhost", "localhost2"]
         !!! This list must have at least 1 element.
         !!! Use machine names within your local network, IP addresses, or
         !!! empty string for the current node running this model.

 forall(n in NodeList) MaxMod(n):= NUMPAR
         !!! Adapt this setting to number of processors and licences per node

 forall(n in NodeList, nct as counter) do
  create(nodeInst(n))
  if connect(nodeInst(n), n)<>0: exit(1)
  if nct>= J then break; end-if     ! Stop if started enough instances
 end-do

!**** Loading model instances ****
 nct:=0
 forall(n in NodeList, m in 1..MaxMod(n), nct as counter) do
   create(modPar(nct))
   load(nodeInst(n), modPar(nct), "rmt:rtparams.bim")  ! Load the bim file
   modPar(nct).uid:= nct            ! Store the model ID as UID
   modNode(modPar(nct).uid):= getsysinfo(nodeInst(n), SYS_NODE)
 end-do


 JobList:= sum(i in JOBS) [i]       ! Define the list of jobs (instances)
 JobSize:=JobList.size              ! Store the number of jobs
 JobsRun:={}                        ! Set of terminated jobs is empty

!**** Start initial lot of model runs ****
 forall(m in RM)
   if JobList<>[] then
     start_next_job(modPar(m))
   end-if

!**** Run all remaining jobs ****
 while (JobsRun.size<JobSize) do
   wait                             ! Wait for model termination
  ! Start next job
   Msg:= getnextevent
   if Msg.class=EVENT_END then      ! We are only interested in "end" events
     m:= Msg.fromuid                ! Retrieve the model UID
     JobsRun+={jobid(m)}            ! Keep track of job termination
     writeln_("End of job ", jobid(m), " (model ", m, ")")
     if JobList<>[] then            ! Start a new run if queue not empty
      start_next_job(modPar(m))
     end-if
   end-if
 end-do

 fdelete("rtparams.bim")            ! Cleaning up
end-model
```

A new function used by this example is `sethostalias`: the elements of the list `NodeList` are used for indexing various arrays and must therefore all be different. If we want to start several Mosel instances on the same machine (here: the local node, designated by `localhost`) we must make sure that they are addressed with different names. These names can be set up through `sethostalias` (see the Mosel Language Reference Manual for further detail).

The procedure `start_next_job` that starts the next job from the queue on remains exactly the same

as before (in Section 2.6.3) and its definition is not repeated here.

2.8.4 Finding available Mosel servers

In the preceding examples we have specified the names of the Mosel instances that we wish to connect to. It is also possible to search for Mosel servers on the local network and hence decide dynamically which remote instances to use for the processing of submodels. This search functionality is provided by the procedure `findxsrvs` that returns a set of at most `M` (second argument) instance names in its third argument. The first subroutine argument selects the group number of the servers, further configuration options such as the port number to be used are accessible through control paramters (see the documentation of *mmjobs* in the Mosel Language Reference for the full list).

```
model "Find Mosel servers"
 uses "mmjobs"

 parameters
   M = 20                  ! Max. number of servers to be sought
 end-parameters

 declarations
   Hosts: set of string
   mosInst: Mosel
 end-declarations

 findxsrvs(1, M, Hosts)
 writeln_(Hosts.size, " servers found: ", Hosts)

 forall(i in Hosts)        ! Establish remote connection and print system info
   if connect(mosInst, i)=0 then
     writeln_("Server ", i, ": ", getsysinfo(mosInst))
     disconnect(mosInst)
   else
     writeln_("Connection to ", i, " failed ")
   end-if
end-model
```

2.9 XPRD: Remote model execution without local installation

In a distributed Mosel application, the local Mosel (main/parent) model that we have seen in the previous section can be replaced by an XPRD (C or Java) program. The *Mosel remote invocation library (XPRD)* makes it possible to build applications requiring the Xpress technology that run from environments where Xpress is not installed—including architectures for which Xpress is not available. XPRD is a self-contained library (*i.e.* with no dependency on the usual Xpress libraries) that provides the necessary routines to start Mosel instances either on the local machine or on remote hosts and control them in a similar way as if they were invoked through the Mosel libraries. Besides the standard instance and model handling operations (connect/disconnect, compile-load-run, stream redirection), the XPRD library supports the file handling mechanisms of *mmjobs* (transparent file access between instances) as well as its event signaling system (events can be exchanged between the application and running models).

The following examples show how to replace the controlling Mosel model in the examples from Section 2.8 by a Java program. The Xpress distribution equally includes C versions of these examples in the subdirectory `examples/mosel/WhitePapers/MoselPar/XPRD`. The Mosel (sub)models remain unchanged from the versions shown in the previous section.

2.9.1 Executing a model remotely

The structure of a Java program for compiling and launching the Mosel model `rtparams.mos` on a remote Mosel instance remains exactly the same as what we have seen for the Mosel model in Section

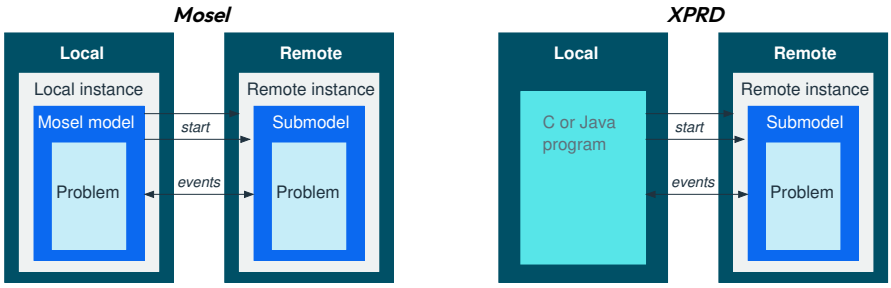


Figure 5: Remote model execution with and without local installation

2.8.1, including the use of the `rmt` driver indicating that the Mosel source is located on a remote machine. The generated BIM file is saved into Mosel's temporary working directory (using the `tmp` driver).

```
import com.dashoptimization.*;
import java.lang.*;
import java.io.*;

public class runrtdistr
{
 public static void main(String[] args) throws Exception
 {
  XPRD xprd=new XPRD();
  XPRDMosel mosInst=null;
  XPRDModel modPar:=null;

       // Use the name or IP address of a machine in
       // your local network, or "" for current node
  String NODENAME = "";
                               // Open connection to a remote node
  mosInst=xprd.connect(NODENAME);
                               // Compile the model file
  mosInst.compile("", "rmt:rtparams.mos", "tmp:rp.bim");
                               // Load bim file into the remote instance
  modPar=mosInst.loadModel("tmp:rp.bim");
                               // Run-time parameters
  modPar.execParams = "PARAM1=" + 2 + ",PARAM2=" + 3.4 +
                      ",PARAM3='a string'" + ",PARAM4=true";
  modPar.run();                // Run the model
  xprd.waitForEvent();         // Wait for model termination
  xprd.dropNextEvent();        // Ignore termination event message

  System.out.println(" `rtparams' returned: " + modPar.getResult());

  mosInst.disconnect();        // Disconnect remote instance
 }
}
```

For compiling and running this Java program only the XPRD library is required, the corresponding commands typically look as follows (Windows version, assuming that `xprd.jar` is located in the current working directory):

```
javac -cp xprd.jar:. runrtdistr.java
java -cp xprd.jar:. runrtdistr
```

Note: In this example, we assume that we know of a specific Mosel instance that we wish to connect to. The program example in Section 2.9.4 shows how to search for available *xprmsrv* servers on the local network.

2.9.2 Parallel models in distributed architecture

The following Java program `runrtpardistr.java` extends the simple example from the previous section to running A=10 model instances on B=5 Mosel instances, assigning 2 models to each instance. All submodels are started concurrently and the Java programm waits for their termination.

```
import com.dashoptimization.*;
import java.lang.*;
import java.io.*;

public class runrtpardistr
{
 static final int A=10;
 static final int B=5;

 public static void main(String[] args) throws Exception
 {
  XPRD xprd=new XPRD();
  XPRDMosel[] mosInst=new XPRDMosel[B];
  XPRDModel[] modPar=new XPRDModel[A];
  String[] NODENAMES=new String[B];
  int i,j;

       // Use the name or IP address of a machine in
       // your local network, or "" for current node
  for(j=0;j<B;j++) NODENAMES[j] = "localhost";

                               // Open connection to remote nodes
  for(j=0;j<B;j++)
    mosInst[j]=xprd.connect(NODENAMES[j]);

  for(j=0;j<B;j++)
    System.out.println("Submodel node: " +
       mosInst[j].getSystemInformation(XPRDMosel.SYS_NODE) + " on " +
       mosInst[j].getSystemInformation(XPRDMosel.SYS_NAME));

                               // Compile the model file on one instance
  mosInst[0].compile("", "rmt:rtparams3.mos", "rmt:rp3.bim");

  for(i=0;i<A;i++)
  {                          // Load the bim file into remote instances
    modPar[i]=mosInst[i%B].loadModel("rmt:rp3.bim");
                               // Run-time parameters
    modPar[i].execParams = "PARAM1=" + i + ",PARAM2=" + (0.1*i) +
                           ",PARAM3='a string " + i + "',PARAM4=" + (i%2==0);
    modPar[i].run();           // Run the model
  }

  for(i=0;i<A;i++)
  {
    xprd.waitForEvent();       // Wait for model termination
    xprd.dropNextEvent();      // Ignore termination event message
  }

  for(j=0;j<B;j++)
    mosInst[j].disconnect();   // Disconnect remote instance

  new File("rp3.bim").delete();   // Cleaning up
 }
}
```

2.9.3 Job queue for parallel execution in a distributed architecture

The following Java program implements a job queue for coordinating the execution of a list of model

runs on several remote Mosel instances, similarly to the Mosel model in Section 2.8.3. Each Mosel instance processes up to `NUMPAR` concurrent submodels. The job queue and the list of terminated jobs are represented by `List` structures. The program closely matches the previous implementation with a controlling Mosel model, including the use of the subroutine `startNextJob` to retrieve the next job from the queue and to start its processing.

```
import com.dashoptimization.*;
import java.lang.*;
import java.util.*;
import java.io.*;

public class runrtparqueued
{
 static final int J=10;             // Number of jobs to run
 static final int NUMPAR=2;         // Number of parallel model executions
                                    // (preferrably <= no. of processors)
 static int[] jobid;
 static int[] modid;
 static String[] modNode;

 public static void main(String[] args) throws Exception
 {
  XPRD xprd=new XPRD();
                            // Use the name or IP address of a machine in
                            // your local network, or "" for current node
  String[] NodeList={"localhost","localhost"};
  final int nbNodes=(NodeList.length<J?NodeList.length:J);
  XPRDMosel[] mosInst=new XPRDMosel[nbNodes];
  int[] MaxMod=new int[nbNodes];
  XPRDModel[] modPar=new XPRDModel[nbNodes*NUMPAR];
  int nct;
  List<Integer> JobList=new ArrayList<Integer>();
  List<Integer> JobsRun=new ArrayList<Integer>();
  int JobSize;
  XPRDEvent event;
  int lastId=0;

  //**** Setting up remote Mosel instances ****
  for(int n=0;n<nbNodes;n++)
  {
   mosInst[n]=xprd.connect(NodeList[n]);
   MaxMod[n]= NUMPAR;
         // Adapt this setting to number of processors and licences per node
  }
                                    // Compile the model file on first node
  mosInst[0].compile("", "rmt:rtparams.mos", "rmt:rtparams.bim");

  //**** Loading model instances ****
  nct=0;
  for(int n=0;(n<nbNodes) && (nct<J);n++)
   for(int m=0;(m<MaxMod[n]) && (nct<J);m++)
   {                                // Load the bim file
    modPar[nct]=mosInst[n].loadModel("rmt:rtparams.bim");
    if(modPar[nct].getNumber()>lastId) lastId=modPar[nct].getNumber();
    nct++;
   }

  jobid=new int[lastId+1];
  modid=new int[lastId+1];
  modNode=new String[lastId+1];
  for(int j=0;j<nct;j++)
  {
   int i=modPar[j].getNumber();
   modid[i]=j;                     // Store the model ID
   modNode[i]=modPar[j].getMosel().getSystemInformation(XPRDMosel.SYS_NODE);
  }
```

```
  for(int i=0;i<J;i++)            // Define the list of jobs (instances)
   JobList.add(Integer.valueOf(i));
  JobSize=JobList.size();         // Store the number of jobs
  JobsRun.clear();                // List of terminated jobs is empty

  //**** Start initial lot of model runs ****
  for(int j=0;j<nct;j++)
   startNextJob(JobList,modPar[j]);

  //**** Run all remaining jobs ****
  while(JobsRun.size()<JobSize)
  {
   xprd.waitForEvent();           // Wait for model termination
   event=xprd.getNextEvent();     // We are only interested in "end" events
   if(event.eventClass==XPRDEvent.EVENT_END)
   {                              // Keep track of job termination
    JobsRun.add(Integer.valueOf(jobid[event.sender.getNumber()]));
    System.out.println("End of job "+ jobid[event.sender.getNumber()] +
                       " (model "+ modid[event.sender.getNumber()]+ ")");
    if(!JobList.isEmpty())        // Start a new run if queue not empty
     startNextJob(JobList,event.sender);
   }
  }

  for(int n=0;n<nbNodes;n++)
   mosInst[n].disconnect();                // Terminate remote instances

  new File("rtparams.bim").delete();       // Cleaning up
 }

//******** Start the next job in a queue ********
 static void startNextJob(List<Integer> jobList,XPRDModel model) throws Exception
 {
  Integer job;
  int i;

  job=jobList.remove(0);      // Retrieve and remove first entry from job list
  i=job.intValue();
  System.out.println("Start job "+ job + " (model " + modid[model.getNumber()]+
                     ") on "+ modNode[model.getNumber()] );
  model.execParams = "PARAM1=" + i + ",PARAM2=" + (0.1*i) +
                     ",PARAM3='a string " + i + "',PARAM4=" + (i%2==0);
  model.run();
  jobid[model.getNumber()]=i;
 }
}
```

2.9.4 Finding available Mosel servers

In the preceding examples we have specified the names of the Mosel instances that we wish to connect to. It is also possible to search for Mosel servers on the local network and hence decide dynamically which remote instances to use for the processing of submodels. This search functionality is provided by the method `findXsrvs` of class `XPRD` that returns a set of at most `M` (second argument) instance names in its third argument. The first argument of this method selects the group number of the servers, further configuration options such as the port number are accessible through control paramters (please refer to the documentation of `findXsrvs` in the XPRD Javadoc for further detail).

```
import com.dashoptimization.*;
import java.lang.*;
import java.io.*;
import java.util.*;

public class findservers
{
```

```
 static final int M=20;

 public static void main(String[] args) throws Exception
 {
  XPRD xprd=new XPRD();
  XPRDMosel mosInst=null;
  Set<String> Hosts=new HashSet<String>();

  xprd.findXsrvs(1, M, Hosts);
  System.out.println(Hosts.size() + " servers found.");

  for(Iterator<String> h=Hosts.iterator(); h.hasNext();)
  {
   String i=h.next();

   try {
    mosInst=xprd.connect(i);     // Open connection to a remote node
                                 // Display system information
    System.out.println("Server " + i + ": " + mosInst.getSystemInformation());
    mosInst.disconnect();        // Disconnect remote instance
   }
   catch(IOException e) {
    System.out.println("Connection to " + i + " failed");
   }
  }
 }
}
```

3 Column generation: solving different models in sequence

The *cutting stock example* we are working with in this section is taken from the 'Mosel User Guide'. The reader is refered to this manual for further detail on the column generation algorithm and its implementation with Mosel.

Column generation algorithms are typically used for solving linear problems with a huge number of variables for which it is not possible to generate explicitly all columns of the problem matrix. Starting with a very restricted set of columns, after each solution of the problem a column generation algorithm adds one or several columns that improve the current solution.

Our column generation algorithm for the cutting stock problem requires us to solve a knapsack problem based on the dual value of the current solution to determine a new column (= cutting pattern). The difference between the User Guide implementation and the one shown below consists in the handling of this knapsack (sub)problem. In the User Guide implementation Mosel's *constraint hiding* functionality is used to blend out subsets of constraints; in the version shown below the subproblem is implemented in a model on its own. Both versions implement exactly the same algorithm and their performance is comparable. On larger instances, however, the two-model version is likely to be slightly more efficient, since every model defines exactly the problem to be solved, without any selection of (un)hidden constraints.

In this example, the changes to the problems are such that they cause complete re-loading of the problems for every optimization run. A clearer advantage of the multi-model version would show up if there were only slight changes (bound updates) to the main (cutting stock) problem so that this problem did not have to be reloaded into the solver for every new run.

3.1 Example problem: cutting stock

A paper mill produces rolls of paper of a fixed width $MAXWIDTH$ that are subsequently cut into smaller rolls according to the customer orders. The rolls can be cut into $NWIDTHS$ different sizes. The orders are given as demands for each width i ($DEMAND_i$). The objective of the paper mill is to satisfy the demand with the smallest possible number of paper rolls in order to minimize the losses.

The objective of minimizing the total number of rolls can be expressed as choosing the best set of cutting patterns for the current set of demands. Since it may not be obvious how to calculate all possible cutting patterns by hand, we start off with a basic set of patterns ($PATTERNS_1$,..., $PATTERNS_{NWIDTH}$), that consists of cutting small rolls all of the same width as many times as possible (and at most the demanded quantity) out of the large roll.

If we define variables use_j to denote the number of times a cutting pattern j ($j \in WIDTHS = \{1, ..., NWIDTH\}$) is used, then the objective becomes to minimize the sum of these variables, subject to the constraints that the demand for every size has to be met.

$$\text{minimize} \sum_{j \in WIDTHS} use_j$$

$$\sum_{j \in WIDTHS} PATTERNS_{ij} \cdot use_j \geqslant DEMAND_i$$

$$\forall j \in WIDTHS : use_j \leqslant \lceil DEMAND_j / PATTERNS_{jj} \rceil, \; use_j \in \mathbf{N}$$

The paper mill can satisfy the demand with just the basic set of cutting patterns, but it is likely to incur significant losses through wasting more than necessary of every large roll and by cutting more small rolls than its customers have ordered. We therefore employ a column generation heuristic to find more suitable cutting patterns.

Our heuristic performs a column generation loop at the top node, before starting the MIP search. Every iteration of the column generation loop executes the following steps:

1. solve the LP and save the basis
2. get the solution values
3. compute a more profitable cutting pattern based on the current solution
4. generate a new column (= cutting pattern): add a term to the objective function and to the corresponding demand constraints
5. load the modified problem and load the saved basis

Step 3 of this loop requires us to solve an *integer knapsack problem* of the form

$$\text{maximize } z = \sum_{j \in WIDTHS} C_j \cdot x_j$$
$$\sum_{j \in WIDTHS} A_j \cdot x_j \leqslant B$$
$$\forall j \in WIDTHS : x_j \text{ integer}$$

This second optimization problem is independent of the main, cutting stock problem since the two have no variables in common.

3.2 Implementation

The implementation is divided into two parts: the *main model* (file `paperp.mos`) with the definition of the cutting stock problem and the column generation algorithm, and the *knapsack model* (file `knapsack.mos`) that is run as a submodel.

3.2.1 Main model

The main part of the cutting stock model looks as follows:

```
model "Papermill (multi-model)"
 uses "mmxprs", "mmjobs"

 forward procedure column_gen
 forward function knapsack(C:array(range) of real,
                           A:array(range) of real,
                           B:real, D:array(range) of integer,
                           xbest:array(range) of integer): real

 declarations
  NWIDTHS = 5                          ! Number of different widths
  WIDTHS = 1..NWIDTHS                  ! Range of widths
  RP: range                            ! Range of cutting patterns
  MAXWIDTH = 94                        ! Maximum roll width
  EPS = 1e-6                           ! Zero tolerance

  WIDTH: array(WIDTHS) of real         ! Possible widths
  DEMAND: array(WIDTHS) of integer     ! Demand per width
  PATTERNS: array(WIDTHS, WIDTHS) of integer  ! (Basic) cutting patterns

  use: dynamic array(RP) of mpvar      ! Rolls per pattern
  soluse: dynamic array(RP) of real    ! Solution values for variables  `use'
  Dem: array(WIDTHS) of linctr         ! Demand constraints
```

```
 MinRolls: linctr                      ! Objective function

 Knapsack: Model                       ! Reference to the knapsack model
end-declarations

WIDTH::  [ 17, 21, 22.5,  24, 29.5]
DEMAND:: [150, 96,   48, 108,  227]

forall(j in WIDTHS)                    ! Make basic patterns
  PATTERNS(j,j) := minlist(floor(MAXWIDTH/WIDTH(j)),DEMAND(j))

forall(j in WIDTHS) do
 create(use(j))                        ! Create NWIDTHS variables  `use'
 use(j) is_integer                     ! Variables are integer and bounded
 use(j) <= integer(ceil(DEMAND(j)/PATTERNS(j,j)))
end-do

MinRolls:= sum(j in WIDTHS) use(j)     ! Objective: minimize no. of rolls

                                       ! Satisfy all demands
forall(i in WIDTHS)
 Dem(i):= sum(j in WIDTHS) PATTERNS(i,j) * use(j) >= DEMAND(i)

res:= compile("knapsack.mos")          ! Compile the knapsack model
load(Knapsack, "knapsack.bim")         ! Load the knapsack model
column_gen                             ! Column generation at top node

minimize(MinRolls)                     ! Compute the best integer solution
                                       ! for the current problem (including
                                       ! the new columns)
writeln_("Best integer solution: ", getobjval, " rolls")
write_("  Rolls per pattern: ")
forall(i in RP) write(getsol(use(i)),", ")
writeln
end-model
```

Before starting the column generation heuristic (the definition of procedure `column_gen` is left out here since it remains unchanged from the User Guide example) the knapsack model is compiled and loaded so that at every column generation loop we merely need to run it with new data. The knapsack model is run from the function `knapsack` that takes as its parameters the data for the knapsack problem and its solution values. The function saves all data to shared memory, then runs the knapsack model and retrieves the solution from shared memory. Its return value is the objective value (`zbest`) of the knapsack problem.

```
function knapsack(C:array(range) of real,
                  A:array(range) of real,
                  B:real, D:array(range) of integer,
                  xbest:array(range) of integer):real

 INDATA  := "bin:shmem:indata"
 RESDATA := "bin:shmem:resdata"

 initializations to INDATA
  A  B  C  D
 end-initializations

 run(Knapsack, "NWIDTHS=" + NWIDTHS + ",INDATA=" + INDATA +
     ",RESDATA=" + RESDATA)            ! Start knapsack (sub)model
 wait                                  ! Wait until subproblem finishes
 dropnextevent                         ! Ignore termination message

 initializations from RESDATA
  xbest  returned as "zbest"
 end-initializations
end-function
```

To enforce a *sequential execution* of the two models (we need to retrieve the results from the knapsack problem before we may continue with the main problem) we must add a call to the procedure `wait` immediately after the `run` statement. Otherwise the execution of the parent model continues concurrently to the child model. On termination, the child model sends a 'termination' event (an event of class `EVENT_END`). Since our algorithm does not require this event we simply remove it from the model's event queue with a call to `dropnextevent`.

3.2.2 Knapsack model

The implementation of the knapsack model is straightforward. All problem data is obtained from shared memory and after solving the problem its solution is saved into shared memory.

```
model "Knapsack"
 uses "mmxprs"

 parameters
  NWIDTHS=5                          ! Number of different widths
  INDATA = "knapsack.dat"            ! Input data file
  RESDATA = "knresult.dat"           ! Result data file
 end-parameters

 declarations
  WIDTHS = 1..NWIDTHS                ! Range of widths
  A,C: array(WIDTHS) of real         ! Constraint + obj. coefficients
  B: real                            ! RHS value of knapsack constraint
  D: array(WIDTHS) of integer        ! Variables bounds (demand quantities)
  KnapCtr, KnapObj: linctr           ! Knapsack constraint+objective
  x: array(WIDTHS) of mpvar          ! Knapsack variables
  xbest: array(WIDTHS) of integer    ! Solution values
 end-declarations

 initializations from INDATA
  A  B  C  D
 end-initializations

! Define the knapsack problem
 KnapCtr:= sum(j in WIDTHS) A(j)*x(j) <= B
 KnapObj:= sum(j in WIDTHS) C(j)*x(j)

 forall(j in WIDTHS) x(j) is_integer
 forall(j in WIDTHS) x(j) <= D(j)

! Solve the problem and retrieve the solution
 maximize(KnapObj)
 z:=getobjval
 forall(j in WIDTHS) xbest(j):=round(getsol(x(j)))

 initializations to RESDATA
  xbest  z as "zbest"
 end-initializations

end-model
```

3.3 Alternative implementation with multiple problems

The solving of the main problem and the knapsack subproblem is sequential. An alternative implementation of the column generation algorithm therefore uses several *problems* (in the place of several *models*). Since the different problems are contained in a single model file, all Mosel code referring to data transfer between models is removed in this version; the mathematical part of the model remains the same. The complete Mosel program is somewhat shorter

3.3.1 Main problem

The main model body now simply defines the cutting stock problem, starts the column generation loop followed by the MIP optimization, and prints out the results, just like the model version documented in the *Mosel User Guide*.

The definition of the column generation algorithm in subroutine `column_gen` remains entirely unchanged and we therefore omit its listing here .

3.3.2 Knapsack problem

The complete formulation of the knapsack problem is now contained in the function `knapsack`, where the submodel is re-defined and re-solved at every call to the function. The line `with mpproblem do` indicates that we switch to a new problem, created locally at this point. After the `do` block we return automatically to the main (cutting stock) problem.

```
function knapsack(C:array(range) of real,
                  A:array(range) of real,
                  B:real, D:array(range) of integer,
                  xbest:array(range) of integer):real

 declarations
  x: array(WIDTHS) of mpvar            ! Knapsack variables
 end-declarations

 with mpproblem do                     ! Create a local subproblem

! Define the knapsack problem
  forall(j in WIDTHS) x(j) is_integer
  forall(j in WIDTHS) x(j) <= D(j)
  sum(j in WIDTHS) A(j)*x(j) <= B

  maximize(sum(j in WIDTHS) C(j)*x(j))
  returned:=getobjval
  forall(j in WIDTHS) xbest(j):=round(getsol(x(j)))

 end-do

end-function
```

Slightly more efficient is the following version where the declaration of the knapsack problem has been moved into the main model body, that is, the problem is declared *globally* instead of being re-created and deleted at every call to the subroutine `knapsack`.

```
declarations
 Knapsack: mpproblem                  ! Knapsack subproblem
 KnapCtr, KnapObj: linctr             ! Knapsack constraint+objective
 x: array(WIDTHS) of mpvar            ! Knapsack variables
end-declarations


function knapsack(C:array(range) of real,
                  A:array(range) of real,
                  B:real, D:array(range) of integer,
                  xbest:array(range) of integer,
                  pass: integer):real

 with Knapsack do

! Redefine the knapsack problem
  KnapCtr := sum(j in WIDTHS) A(j)*x(j) <= B
  KnapObj := sum(j in WIDTHS) C(j)*x(j)
```

```
! Integrality condition
  if pass=1 then
   forall(j in WIDTHS) x(j) is_integer
   forall(j in WIDTHS) x(j) <= D(j)
  end-if

  maximize(KnapObj)
  returned:=getobjval
  forall(j in WIDTHS) xbest(j):=round(getsol(x(j)))

 end-do

end-function
```

3.4 Results

With the data in the model above, the column generation algorithm generates 6 new patterns, taking the value of the LP-relaxation of the cutting stock problem from originally 177.67 down to 160.95. The MIP finds a solution with 161 rolls using the following patterns:

	Widths					
Pattern	**17**	**21**	**22.5**	**24**	**29.5**	**Usage**
3	0	0	4	0	0	1
5	0	0	0	0	3	15
6	0	1	0	3	0	32
8	2	0	0	0	2	75
10	0	2	1	0	1	32
11	0	0	2	2	0	6

4 Solving several model instances in parallel

In this section we show how to execute several models in parallel and communicate solution information among these models. This scheme may be particularly interesting when working with Mosel on a multi-processor machine, *e.g.* by starting a number of models that corresponds to the available number of processors.

Our idea is to run several instances (different only by the parameterization of the solution algorithm) of the same MIP model concurrently and to stop the entire run when the first model has finished. If the different solution algorithms are complementary in the sense that some quickly produce (good) solutions and others are better at proving optimality once the best solution is found then one may reasonably expect an additional synergy effect from exchanging solution updates during the MIP search.

To implement this scheme, we define a main model that starts the model runs and coordinates the solution updates, and a parameterizable child model that is loaded and run with the desired number of versions. The child models all use the same solver (Xpress Optimizer) but it would equally be possible to use a different solver for some of the child models, provided it defines the necessary functionality for interacting with the search.

4.1 Example problem: economic lot sizing

Economic lot sizing (ELS) considers production planning over a given planning horizon, in our example a range of time periods $\mathrm{TIMES} = 1, \ldots, T$. In every period t, there is a given demand DEMAND_{pt} for every product p ($p \in \mathrm{PRODUCTS}$) that must be satisfied by the production in this period and by inventory carried over from previous periods.

A set-up cost $\mathrm{SETUPCOST}_t$ is associated with production in a period, and the total production capacity per period, CAP_t, is limited. The unit production cost $\mathrm{PRODCOST}_{pt}$ per product and time period is also given. There is no inventory or stock-holding cost.

We introduce the decision variables $\mathrm{produce}_{pt}$ for the amount of product p made in period t and the binary variables setup_{pt} indicating whether a setup takes place for product p in period t ($\mathrm{setup}_{pt} = 1$) or not ($\mathrm{setup}_{pt} = 0$).

We may then formulate the following mathematical model for this problem:

$$\text{minimize} \sum_{t \in \mathrm{TIMES}} \left(\mathrm{SETUPCOST}_t \cdot \sum_{p \in \mathrm{PRODUCTS}} \mathrm{setup}_{pt} + \sum_{p \in \mathrm{PRODUCTS}} \mathrm{PRODCOST}_{pt} \cdot \mathrm{produce}_{pt} \right)$$

$$\forall p \in \mathrm{PRODUCTS}, t \in \mathrm{TIMES} : \sum_{s=1}^{t} \mathrm{produce}_{ps} \geqslant \sum_{s=1}^{t} \mathrm{DEMAND}_{ps}$$

$$\forall p \in \mathrm{PRODUCTS}, t \in \mathrm{TIMES} : \mathrm{produce}_{pt} \leqslant D_{ptT} \cdot \mathrm{setup}_{pt}$$

$$\forall t \in \mathrm{TIMES} : \sum_{p \in \mathrm{PRODUCTS}} \mathrm{produce}_{pt} \leqslant \mathrm{CAP}_t$$

$$\forall p \in \mathrm{PRODUCTS}, t \in \mathrm{TIMES} : \mathrm{setup}_{pt} \in \{0, 1\}, \mathrm{produce}_{pt} \geqslant 0$$

The objective function is to minimize the total cost. The constraints in the second line formulate the requirement that the production of p in periods 0 to t must satisfy the total demand for this product during this period of time. The next set of constraints establish the implication 'if there is production during t then there is a setup in t' where D_{ptl} stands for the demand of product p in periods t to l. The production capacity per period t is limited. And finally, the setup_{pt} variables are binaries.

4.1.1 Cutting plane algorithm

A well-known class of valid inequalities for ELS are the so-called (l, S)-inequalities [**?**]. If D_{ptl} denotes the total demand of p in periods t to l, then for each period l and each subset of periods S of 1 to l, the (l, S)-inequality is

$$\sum_{\substack{t=1 \\ t \in S}}^{l} produce_{pt} + \sum_{\substack{t=1 \\ t \notin S}}^{l} D_{ptl} \cdot setup_{pt} \geqslant D_{pll}$$

It says that actual production $produce_{pt}$ in the periods included in S plus the maximum potential production $D_{ptl} \cdot setup_{pt}$ in the remaining periods (those not in S) must at least equal the total demand in periods 1 to l.

It is possible to develop the following cutting plane algorithm based on these (l, S)-inequalities:

1. Solve the LP.
2. Identify violated (l, S)-inequalities by testing violations of

$$\sum_{t=1}^{l} \min(produce_{pt}, D_{ptl} \cdot setup_{pt}) \geqslant D_{pll}$$

3. Add violated inequalities as cuts to the problem.
4. Re-solve the LP problem.

There are numerous options for how to configure this algorithm. For instance:

- Generation of cuts only in the root node or also during the search (Cut-and-Branch versus Branch-and-Cut).
- Number of cut generation passes at a node (*e.g.* one pass or looping around steps 2.-4. until no more cuts are generated).
- Search tree depth for cut generation (up to a given depth or at all nodes).
- Exclusive use of (l, S)-cuts or combination with others (*e.g.* default cuts generated by the solver).

The implementation of the (l, S)-cut generation algorithm shown below may be configured to generate cuts at the top node only (`TOPONLY = true`) and to generate one or several rounds of cuts (`SEVERALROUNDS = true`).

4.2 Implementation

With *mmjobs* events are always sent between parent – child pairs, not directly from one child to another. The 'solution found' message therefore needs to be sent to the parent model that then passes on this message to all other child models.

Another point that should be stressed is the fact that we only compile the ELS model file once, but the number of instances loaded into memory needs to correspond to the number of child models we wish to run.

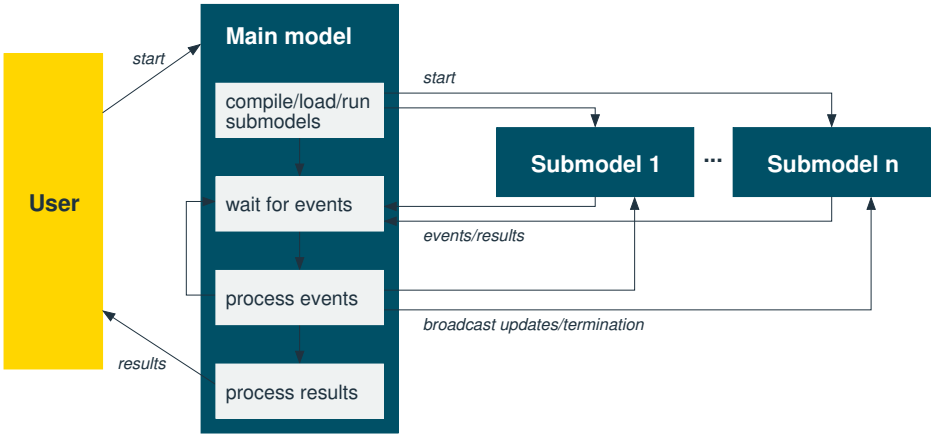


Figure 6: Parallel submodels coordinated by events

4.2.1 Main model

The main model compiles, loads and runs the child models and coordinates the solution updates. Some care must be taken with the solution updates since new solutions that are reported are not guaranteed to be better than others previously reported by other child models. For instance, if two models find solutions almost at the same time, the first solution that reaches the parent may be the better one and it must not be overridden by the next.

For a nice solution display at the end, the main model also reads in parts of the problem data from file.

```
model "Els main"
 uses "mmjobs"

 parameters
  DATAFILE = "els5.dat"
  T = 45
  P = 4
 end-parameters

 declarations
  RM = 0..5                                ! Range of models
  TIMES = 1..T                             ! Time periods
  PRODUCTS = 1..P                          ! Set of products
  solprod: array(PRODUCTS,TIMES) of real   ! Sol. values for var.s produce
  solsetup: array(TIMES) of real           ! Sol. values for var.s setup
  DEMAND: array(PRODUCTS,TIMES) of integer ! Demand per period

  modELS: array(RM) of Model               ! Models
  NEWSOL = 2                               ! Identifier for "sol. found" event
  Msg: Event                               ! Messages sent by models
  params: text                             ! Submodel runtime parameters
 end-declarations

! Compile, load, and run models M1 and M2
 M1:= 1; M2:=2
 res:= compile("elsp.mos")                 ! Compile the submodel
 load(modELS(M1), "elsp.bim")              ! Load submodels
 load(modELS(M2), "elsp.bim")
 forall(m in RM) modELS(m).uid:= m         ! Store the model indices
 setmodpar(params, "DATAFILE", DATAFILE)   ! Configure submodel parameters
 setmodpar(params, "T", T); setmodpar(params, "P", P)
                                           ! Start submodel runs
 setmodpar(params, "ALG", M1); run(modELS(M1), params)
 setmodpar(params, "ALG", M2); run(modELS(M2), params)
```

```
 objval:= MAX_REAL
 algsol:= -1; algopt:= -1

 repeat
  wait                                ! Wait for the next event
  Msg:= getnextevent                  ! Get the event
  if getclass(Msg)=NEWSOL then        ! Get the event class
   if getvalue(Msg) < objval then     ! Value of the event (= obj. value)
    algsol:= getfromuid(Msg)          ! UID of model sending the event
    objval:= getvalue(Msg)
    writeln_("Improved solution ", objval, " found by model ", algsol)
    forall(m in RM | m <> algsol) send(modELS(m), NEWSOL, objval)
   else
    writeln_("Solution ", getvalue(Msg), " found by model ", Msg.fromuid)
   end-if
  end-if
 until getclass(Msg)=EVENT_END        ! A model has finished

 algopt:= getfromuid(Msg)             ! Retrieve ID of terminated model
 forall(m in RM) stop(modELS(m))      ! Stop all running models

 if algsol=-1 then
  writeln_("No solution available")
  exit(1)
 else                    ! Retrieve the best solution from shared memory
  initializations from "bin:shmem:sol"+algsol
   solprod
   solsetup
  end-initializations

  initializations from DATAFILE
   DEMAND
  end-initializations

! Solution printing
  writeln_("Best solution found by model ", algsol)
  writeln_("Optimality proven by model ", algopt)
  writeln_("Objective value: ", objval)
  write_("Period  setup    ")
  forall(p in PRODUCTS) write(strfmt(p,-7))
  forall(t in TIMES) do
   write("\n ", strfmt(t,2), strfmt(solsetup(t),8), "    ")
   forall(p in PRODUCTS) write(strfmt(solprod(p,t),3), " (",DEMAND(p,t),")")
  end-do
  writeln
 end-if

end-model
```

In this implementation we save the model index used in our model as UID (user index) information with each model object. Whenever a child model sends an event to the parent model, we retrieve its UID (with function `getfromuid`) to identify the corresponding submodel in our model and use this information for solution printing later on.

4.2.2 ELS model

The ELS child model is written in such a way that the model can be executed separately. In particular, every model performs the complete initialization of its data from file, a task that for greater efficiency could be reserved to the parent model, communicating data via shared memory to the child models (however, in our example data handling time is negligible compared to the running time of the solution algorithms).

The main part of the ELS model contains the definition of the model itself and the selection of the

solution algorithm:

```
model Els
 uses "mmxprs","mmjobs"

 parameters
  ALG = 0                                  ! Default algorithm: no user cuts
  DATAFILE = "els4.dat"
  T = 60
  P = 4
 end-parameters

 forward procedure tree_cut_gen
 forward function cb_node: boolean
 forward function cb_updatebnd: boolean
 forward procedure cb_intsol

 declarations
  NEWSOL = 2                               ! "New solution" event class
  EPS = 1e-6                               ! Zero tolerance
  TIMES = 1..T                             ! Time periods
  PRODUCTS = 1..P                          ! Set of products

  DEMAND: array(PRODUCTS,TIMES) of integer ! Demand per period
  SETUPCOST: array(TIMES) of integer       ! Setup cost per period
  PRODCOST: array(PRODUCTS,TIMES) of real  ! Production cost per period
  CAP: array(TIMES) of integer             ! Production capacity per period
  D: array(PRODUCTS,TIMES,TIMES) of integer ! Total demand in periods t1 - t2

  produce: array(PRODUCTS,TIMES) of mpvar  ! Production in period t
  setup: array(TIMES) of mpvar             ! Setup in period t

  solprod: array(PRODUCTS,TIMES) of real   ! Sol. values for var.s produce
  solsetup: array(TIMES) of real           ! Sol. values for var.s setup

  Msg: Event                               ! An event
 end-declarations

 initializations from DATAFILE
  DEMAND SETUPCOST PRODCOST CAP
 end-initializations

 forall(p in PRODUCTS,s,t in TIMES) D(p,s,t):= sum(k in s..t) DEMAND(p,k)

! Objective: minimize total cost
 MinCost:= sum(t in TIMES) (SETUPCOST(t) * setup(t) +
                            sum(p in PRODUCTS) PRODCOST(p,t) * produce(p,t) )

! Production in period t must not exceed the total demand for the
! remaining periods; if there is production during t then there
! is a setup in t
 forall(t in TIMES)
  sum(p in PRODUCTS) produce(p,t) <= sum(p in PRODUCTS) D(p,t,T) * setup(t)

! Production in periods 0 to t must satisfy the total demand
! during this period of time
 forall(p in PRODUCTS,t in TIMES)
   sum(s in 1..t) produce(p,s) >= sum (s in 1..t) DEMAND(p,s)

! Capacity limits
 forall(t in TIMES) sum(p in PRODUCTS) produce(p,t) <= CAP(t)

 forall(t in TIMES) setup(t) is_binary     ! Variables setup are 0/1

 setparam("zerotol", EPS/100)              ! Set Mosel comparison tolerance
 SEVERALROUNDS:=false; TOPONLY:=false
```

```
 case ALG of
  1: do
      setparam("XPRS_CUTSTRATEGY", 0)      ! No cuts
      setparam("XPRS_HEUREMPHASIS", 0)     ! No heuristics
     end-do
  2: do
      setparam("XPRS_CUTSTRATEGY", 0)      ! No cuts
      setparam("XPRS_HEUREMPHASIS", 0)     ! No heuristics
      setparam("XPRS_PRESOLVE", 0)         ! No presolve
     end-do
  3: tree_cut_gen                          ! User branch&cut (single round)
  4: do                                    ! User branch&cut (several rounds)
      tree_cut_gen
      SEVERALROUNDS:=true
     end-do
  5: do                                    ! User cut&branch (several rounds)
      tree_cut_gen
      SEVERALROUNDS:=true
      TOPONLY:=true
     end-do
 end-case

! Parallel setup
 setcallback(XPRS_CB_PRENODE, ->cb_updatebnd) ! Node pre-treatment callback
 setcallback(XPRS_CB_INTSOL, ->cb_intsol)  ! Integer solution callback

! Solve the problem
 minimize(MinCost)

end-model
```

The procedure `tree_cut_gen` sets up a user cut generation routine, configurable to generate cuts only at the top node of the branch-and-bound search (`TOPONLY`) or to execute one or several cut generation iterations per node (`SEVERALROUNDS`). The definition of the cut generation routine `cb_node` itself is left out here.

```
 procedure tree_cut_gen
  setparam("XPRS_HEUREMPHASIS", 0)         ! Switch heuristics off
  setparam("XPRS_CUTSTRATEGY", 0)          ! Switch automatic cuts off
  setparam("XPRS_PRESOLVE", 0)             ! Switch presolve off
  setparam("XPRS_EXTRAROWS", 5000)         ! Reserve extra rows in matrix

  setcallback(XPRS_CB_OPTNODE, ->cb_node)  ! Define the optnode callback
 end-procedure
```

The communication between concurrently running child models has two parts: (a) any integer solution found must be saved and comunicated to the parent model and (b) bound updates sent by the controlling model must be incorporated into the search. Xpress Optimizer provides a specific *integer solution callback* for saving solutions into user structures. An obvious place for bound updates in nodes is the *cut-manager callback* function. However, this function being already in use for other purposes with certain settings of the algorithm, we employ a different callback function that also gets called at every node, the *node pre-treatment callback*.

```
! **** Update cutoff value ****
 function cb_updatebnd: boolean
  if not isqueueempty then
   repeat
    Msg:= getnextevent
   until isqueueempty
   newcutoff:= getvalue(Msg)
   if newcutoff < getparam("XPRS_MIPABSCUTOFF") then
    setparam("XPRS_MIPABSCUTOFF", newcutoff)
   end-if
```

```
   if (newcutoff < getparam("XPRS_LPOBJVAL")) then
    returned:= true                        ! Node becomes infeasible
   end-if
  end-if
 end-function

! **** Store and communicate new solution ****
 procedure cb_intsol
  objval:= getparam("XPRS_LPOBJVAL")       ! Retrieve current objective value
  cutoff:= getparam("XPRS_MIPABSCUTOFF")
  if(cutoff > objval) then
   setparam("XPRS_MIPABSCUTOFF", objval)
  end-if

 ! Get the solution values
  forall(t in TIMES) do
   forall(p in PRODUCTS) solprod(p,t):=getsol(produce(p,t))
   solsetup(t):=getsol(setup(t))
  end-do

 ! Store the solution in shared memory
  initializations to "bin:shmem:sol"+ALG
   solprod
   solsetup
  end-initializations

 ! Send "solution found" signal
  send(NEWSOL, objval)
 end-procedure
```

The bound update callback function checks whether the event queue contains any events, if this is the case, it takes all events from the queue and sets the value of the last event as the new cutoff value. The rationale behind the loop for emptying the event queue is that the parent model may have sent several improved solution values since the last check, the best value is always the one sent last, that is, the last in the queue.

The integer solution callback writes the solution values to shared memory, adding the identifier of the model (= value of `ALG`). The latter ensures that two child models that possibly write out their solution at the same time do not use the same memory area.

4.3 Results

A run with two models may generate a log similar to the following one (note that the model that terminates the search is not the same that has found the optimal solution).

```
Improved solution 1283 found by model 2
Improved solution 1250 found by model 2
Improved solution 1242 found by model 1
Improved solution 1236 found by model 2
Improved solution 1234 found by model 2
Best solution found by model 2
Optimality proven by model 1
Objective value: 1234
```

5 Dantzig-Wolfe decomposition: combining sequential and parallel solving

Dantzig-Wolfe decomposition (see [**?**] for further detail) is a solution method for problems where, if a relatively small number of constraints were removed, the problem would fall apart into a number of independent problems. This means, it is possible to re-order the rows and columns of the constraint matrix as shown in Figure 7, where non-zero coefficients only occur within the gray shaded areas. Such a *primal block angular structure* may become immediately apparent by visualizing a problem matrix. However, in most cases it will be necessary to re-organize the constraint definitions, grouping them by common index (sub)sets such as time periods, products, plant locations, and so on.

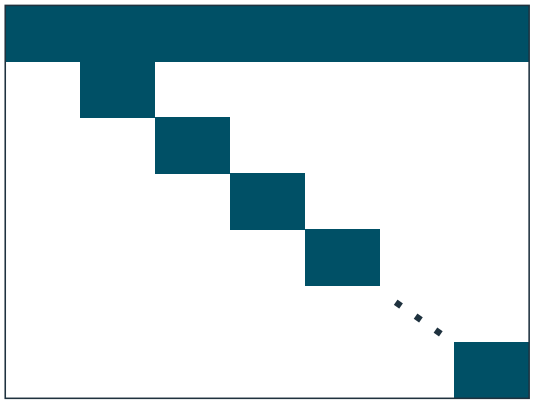


Figure 7: Coeffcient matrix with primal block angular structure

The constraints (including the objective function) involving variables of several or all subproblems are referred to as *global constraints* (also: common or linking constraints). These constraints are used to form the *main problem*. The individual blocks on the diagonal of the coefficient matrix are solved as *pricing subproblems*, coordinated by the main problem. By solving the main problem we obtain a solution to the original problem. Since the main problem has a large number of variables (defined by the set of basic feasible solutions and unbounded directions of the pricing problems), we work with a *restricted main problem* over a small subset of the variables. The variables to enter the active set of variables of the restricted main problem are determined by solving the pricing subproblems. With objective functions for the pricing problems that are based on the dual values of the restricted main problem we can obtain that the objective function value at each extreme point is the reduced cost (or *price*) of the variable of the main problem associated with the extreme point.

For maximization problems, solving the modified pricing problems generates basic feasible solutions of maximum reduced cost. If the objective value at an extreme point is positive, then the associated main problem variable is added to the main problem; if the minimum objective value over all extreme points is negative, then no main problem variable exists to improve the current main problem solution.

The computational advantage of Dantzig-Wolfe decomposition arises from performing a significant amount of the computational work in the pricing problems that are roughly an order of magnitude smaller than the original problem and thus easier to solve. An aspect of the method that is of interest in the context of this paper is that the subproblems are independent of each other, and may therefore be solved concurrently.

A potential drawback of the decomposition approach is the huge size of the main problem, it has many more variables—though fewer constraints—than the original problem. In general it is not required to generate all variables explicitly but since the feasible region of the main problem is more complex than that of the original problem, the solution path may be longer. Furthermore, numerical problems may

occur through the dynamic generation of variables of the main problem.

Many factors may influence the performance of a decomposition approach, so for a particular application computational experiments will be required to find out whether this solution method is suitable. Such tests may include different ways of decomposing a given problem. As a general rule for the definition of a problem decomposition, one should aim for few global constraints since it is important to be able to (re)solve the main problem quickly. In addition, the pricing problems should be constructed in such a way that they are well formed problems in their own right to avoid computational problems due to degeneracy.

5.1 Example problem: multi-item, multi-period production planning

The company Coco has two plants that can produce two types of cocoa powder. The first factory has a total capacity of 400 tons per month and the second of 500 tons per month. The marketing department has provided estimations for the maximum amount of every product that may be sold in each of the next four months, and also the expected sales prices. Several raw materials are used in the production with known raw material requirements per ton of finished products. Finished products and raw material may be stored at the factories from one tim period to the next, incurring a given cost per ton and per time period. The raw material storage capacity at the factories is limited to 300 tons. How should the two plants be operated during the planning period to maximize the total profit?

5.1.1 Original model

Let PRODS be the set of finished products, FACT the set of factories, RAW the set of raw materials, and $\mathrm{PERIODS} = \{1, ..., \mathrm{NT}\}$ the set of time periods under consideration.

We define decision variables $\mathrm{make}_{\mathrm{pft}}$ representing the quantity of product p made at factory f during time period t. Furthermore, to model the transition from one time period to the next and to account for the different types of cost incurred, we need several other sets of variables: $\mathrm{sell}_{\mathrm{pft}}$, the amount of product p sold at factory f in period t; $\mathrm{buy}_{\mathrm{rft}}$ the amount of raw material r bought at f in t; and finally $\mathrm{pstock}_{\mathrm{pft}}$ and $\mathrm{rstock}_{\mathrm{rft}}$ (both defined for $\mathrm{t} = 1, ..., \mathrm{NT} + 1$) respectively the amount of product and raw material held in stock at factory f at the beginning of period t.

Let further $\mathrm{MXSELL}_{\mathrm{pt}}$ be the maximum sales quantity of product p in period t, $\mathrm{MXMAKE}_{\mathrm{f}}$ the capacity limit of factory f, and $\mathrm{MXRSTOCK}$ the raw material storage capacity.

Let $\mathrm{IPSTOCK}_{\mathrm{pf}}$ be the quantity of product p initially held in stock at factory f and $\mathrm{IRSTOCK}_{\mathrm{rf}}$ the initial stock level of raw material r. We denote by $\mathrm{CPSTOCK}$ and $\mathrm{CRSTOCK}$ respectively the unit cost for storing finished product and raw material.

The objective function of maximizing the total profit is to maximize the sales revenues ($\mathrm{REV}_{\mathrm{p}}$), minus the cost of production ($\mathrm{CMAKE}_{\mathrm{pf}}$), buying raw material ($\mathrm{CBUY}_{\mathrm{rt}}$), and storing finished products and raw material.

$$
\begin{aligned}
\text{maximize} \quad & \sum_{\mathrm{p}\in\mathrm{PRODS}} \sum_{\mathrm{f}\in\mathrm{FACT}} \sum_{\mathrm{t}\in\mathrm{PERIODS}} \mathrm{REV}_{\mathrm{pt}} \cdot \mathrm{sell}_{\mathrm{pft}} \\
& - \sum_{\mathrm{p}\in\mathrm{PRODS}} \sum_{\mathrm{f}\in\mathrm{FACT}} \sum_{\mathrm{t}\in\mathrm{PERIODS}} \mathrm{CMAKE}_{\mathrm{pf}} \cdot \mathrm{make}_{\mathrm{pft}} - \sum_{\mathrm{r}\in\mathrm{RAW}} \sum_{\mathrm{f}\in\mathrm{FACT}} \sum_{\mathrm{t}\in\mathrm{PERIODS}} \mathrm{CBUY}_{\mathrm{rt}} \cdot \mathrm{buy}_{\mathrm{rft}} \\
& - \sum_{\mathrm{p}\in\mathrm{PRODS}} \sum_{\mathrm{f}\in\mathrm{FACT}} \sum_{\mathrm{t}=2}^{\mathrm{NT}+1} \mathrm{CPSTOCK} \cdot \mathrm{pstock}_{\mathrm{pft}} - \sum_{\mathrm{r}\in\mathrm{RAW}} \sum_{\mathrm{f}\in\mathrm{FACT}} \sum_{\mathrm{t}=2}^{\mathrm{NT}+1} \mathrm{CRSTOCK} \cdot \mathrm{rstock}_{\mathrm{rft}}
\end{aligned}
$$

The possibility to store products between time periods gives rise to three sets of constraints: the *inventory balance* constraints for finished products ($\mathrm{PBal}_{\mathrm{pft}}$) and for raw material ($\mathrm{RBal}_{\mathrm{rft}}$), and the limit on the raw material storage capacity ($\mathrm{MxRStock}_{\mathrm{ft}}$). The stock $\mathrm{pstock}_{\mathrm{p,f,t+1}}$ of product p at the beginning of $\mathrm{t}+1$ is given by the stock at the beginning of t plus the production in t reduced by the amount sold on t. The stock $\mathrm{rstock}_{\mathrm{r,f,t+1}}$ of raw material r at the beginning of $\mathrm{t}+1$ is given by the corresponding stock at the beginning of t plus the amount bought in t reduced by the quantity used in production during t.

$$\forall \mathrm{p} \in \mathrm{PRODS}, \forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \mathrm{PERIODS} : \mathrm{PBal}_{\mathrm{pft}} := \mathrm{pstock}_{\mathrm{p,f,t+1}} = \mathrm{pstock}_{\mathrm{pft}} + \mathrm{make}_{\mathrm{pft}} - \mathrm{sell}_{\mathrm{pft}}$$

$$\forall \mathrm{r} \in \mathit{RAW}, \forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \mathrm{PERIODS} :$$

$$\mathrm{RBal}_{\mathrm{rft}} := \mathrm{rstock}_{\mathrm{r,f,t+1}} = \mathrm{rstock}_{\mathrm{rft}} + \mathrm{buy}_{\mathrm{rft}} - \sum_{\mathrm{p} \in \mathrm{PRODS}} \mathrm{REQ}_{\mathrm{pr}} \cdot \mathrm{make}_{\mathrm{pft}}$$

$$\forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \{2, ..., \mathrm{NT}+1\} : \mathrm{MxRStock}_{\mathrm{ft}} := \sum_{\mathrm{r} \in \mathit{RAW}} \mathrm{rstock}_{\mathrm{rft}} \leqslant \mathrm{MXRSTOCK}$$

We further have two sets of capacity constraints: the production capacity of the factories is limited (constraints $\mathrm{MxMake}_{\mathrm{ft}}$) and we can only sell up to a given maximum amount of finished products per time period (constraints $\mathrm{MxSell}_{\mathrm{ft}}$).

Below the complete mathematical model is given. The initial stock levels ($\mathrm{t} = 1$) of finished products and raw material are fixed to the given values.

$$\begin{aligned}
\text{maximize } & \sum_{\mathrm{p} \in \mathrm{PRODS}} \sum_{\mathrm{f} \in \mathrm{FACT}} \sum_{\mathrm{t} \in \mathrm{PERIODS}} \mathrm{REV}_{\mathrm{pt}} \cdot \mathrm{sell}_{\mathrm{pft}} \\
& - \sum_{\mathrm{p} \in \mathrm{PRODS}} \sum_{\mathrm{f} \in \mathrm{FACT}} \sum_{\mathrm{t} \in \mathrm{PERIODS}} \mathrm{CMAKE}_{\mathrm{pf}} \cdot \mathrm{make}_{\mathrm{pft}} - \sum_{\mathrm{r} \in \mathit{RAW}} \sum_{\mathrm{f} \in \mathrm{FACT}} \sum_{\mathrm{t} \in \mathrm{PERIODS}} \mathrm{CBUY}_{\mathrm{rt}} \cdot \mathrm{buy}_{\mathrm{rft}} \\
& - \sum_{\mathrm{p} \in \mathrm{PRODS}} \sum_{\mathrm{f} \in \mathrm{FACT}} \sum_{\mathrm{t}=2}^{\mathrm{NT}+1} \mathrm{CPSTOCK} \cdot \mathrm{pstock}_{\mathrm{pft}} - \sum_{\mathrm{r} \in \mathit{RAW}} \sum_{\mathrm{f} \in \mathrm{FACT}} \sum_{\mathrm{t}=2}^{\mathrm{NT}+1} \mathrm{CRSTOCK} \cdot \mathrm{rstock}_{\mathrm{rft}}
\end{aligned}$$

$$\forall \mathrm{p} \in \mathrm{PRODS}, \forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \mathrm{PERIODS} : \mathrm{PBal}_{\mathrm{pft}} := \mathrm{pstock}_{\mathrm{p,f,t+1}} = \mathrm{pstock}_{\mathrm{pft}} + \mathrm{make}_{\mathrm{pft}} - \mathrm{sell}_{\mathrm{pft}}$$

$$\forall \mathrm{r} \in \mathit{RAW}, \forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \mathrm{PERIODS} :$$

$$\mathrm{RBal}_{\mathrm{rft}} := \mathrm{rstock}_{\mathrm{r,f,t+1}} = \mathrm{rstock}_{\mathrm{rft}} + \mathrm{buy}_{\mathrm{rft}} - \sum_{\mathrm{p} \in \mathrm{PRODS}} \mathrm{REQ}_{\mathrm{pr}} \cdot \mathrm{make}_{\mathrm{pft}}$$

$$\forall \mathrm{p} \in \mathrm{PRODS}, \forall \mathrm{t} \in \mathrm{PERIODS} : \mathrm{MxSell}_{\mathrm{pt}} := \sum_{\mathrm{f} \in \mathrm{FACT}} \mathrm{sell}_{\mathrm{pft}} \leqslant \mathrm{MXSELL}_{\mathrm{p}}$$

$$\forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \mathrm{PERIODS} : \mathrm{MxMake}_{\mathrm{ft}} := \sum_{\mathrm{p} \in \mathrm{PRODS}} \mathrm{make}_{\mathrm{pft}} \leqslant \mathrm{MXMAKE}_{\mathrm{f}}$$

$$\forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \{2, ..., \mathrm{NT}+1\} : \mathrm{MxRStock}_{\mathrm{ft}} := \sum_{\mathrm{r} \in \mathit{RAW}} \mathrm{rstock}_{\mathrm{rft}} \leqslant \mathrm{MXRSTOCK}$$

$$\forall \mathrm{p} \in \mathrm{PRODS}, \forall \mathrm{f} \in \mathrm{FACT} : \mathrm{pstock}_{\mathrm{pf1}} = \mathrm{IPSTOCK}_{\mathrm{pf}}$$

$$\forall \mathrm{r} \in \mathit{RAW}, \forall \mathrm{f} \in \mathrm{FACT} : \mathrm{rstock}_{\mathrm{rf1}} = \mathrm{IRSTOCK}_{\mathrm{rf}}$$

$$\forall \mathrm{p} \in \mathrm{PRODS}, \forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \mathrm{PERIODS} : \mathrm{make}_{\mathrm{pft}} \geqslant 0, \mathrm{sell}_{\mathrm{pft}} \geqslant 0$$

$$\forall \mathrm{r} \in \mathit{RAW}, \forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \mathrm{PERIODS} : \mathrm{buy}_{\mathrm{rft}} \geqslant 0$$

$$\forall \mathrm{p} \in \mathrm{PRODS}, \forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \{1, ..., \mathrm{NT}+1\} : \mathrm{pstock}_{\mathrm{pft}} \geqslant 0$$

$$\forall \mathrm{r} \in \mathit{RAW}, \forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{t} \in \{1, ..., \mathrm{NT}+1\} : \mathrm{rstock}_{\mathrm{rft}} \geqslant 0$$

5.1.2 Problem decomposition

We now decompose the problem described above according to production locations. Notice that this is

not the only way of decomposing this problem: we may just as well choose a decomposition by products or by time periods. However, both of these choices result in a larger number of global constraints than the decomposition by factories, meaning that the main problem may become more difficult to solve.

For every factory f we obtain the following subproblem (including the sales limit constraints $MxSell$ in the form of bounds in the submodels is not required, but may lead to better, that is, more exact or faster solutions).

$$\begin{aligned}
\text{maximize } & \sum_{p\in PRODS}\sum_{t\in PERIODS} REV_{pt}\cdot sell_{pt} \\
& - \sum_{p\in PRODS}\sum_{t\in PERIODS} CMAKE_p\cdot make_{pt} - \sum_{r\in RAW}\sum_{t\in PERIODS} CBUY_{rt}\cdot buy_{rt} \\
& - \sum_{p\in PRODS}\sum_{t=2}^{NT+1} CPSTOCK\cdot pstock_{pt} - \sum_{r\in RAW}\sum_{t=2}^{NT+1} CRSTOCK\cdot rstock_{rt} \\
& \forall p\in PRODS, \forall t\in PERIODS : PBal_{pt} := pstock_{p,t+1} = pstock_{pt} + make_{pt} - sell_{pt} \\
& \forall r\in RAW, \forall t\in PERIODS : RBal_{rt} := rstock_{r,t+1} = rstock_{rt} + buy_{rt} - \sum_{p\in PRODS} REQ_{pr}\cdot make_{pt} \\
& \forall t\in PERIODS : MxMake_t := \sum_{p\in PRODS} make_{pt} \leqslant MXMAKE \\
& \forall t\in\{2,\ldots,NT+1\} : MxRStock_t := \sum_{r\in RAW} rstock_{rt} \leqslant MXRSTOCK \\
& \forall p\in PRODS, \forall t\in PERIODS : MxSell_{pt} := sell_{pt} \leqslant MXSELL_p \\
& \forall p\in PRODS : pstock_{p1} = IPSTOCK_p \\
& \forall r\in RAW : rstock_{r1} = IRSTOCK_r \\
& \forall p\in PRODS, \forall t\in PERIODS : make_{pt}\geqslant 0, sell_{pt}\geqslant 0 \\
& \forall r\in RAW, \forall t\in PERIODS : buy_{rt}\geqslant 0 \\
& \forall p\in PRODS, \forall t\in\{1,\ldots,NT+1\} : pstock_{pt}\geqslant 0 \\
& \forall r\in RAW, \forall t\in\{1,\ldots,NT+1\} : rstock_{rt}\geqslant 0
\end{aligned}$$

The main problem only contains a single set of global constraints, namely the sales limit constraints $MxSell$ (for clarity's sake, the non-negativity constraints are left out here).

$$\begin{aligned}
\text{maximize } & \sum_{p\in PRODS}\sum_{f\in FACT}\sum_{t\in PERIODS} REV_{pt}\cdot sell_{pft} \\
& - \sum_{p\in PRODS}\sum_{f\in FACT}\sum_{t\in PERIODS} CMAKE_{pf}\cdot make_{pft} - \sum_{r\in RAW}\sum_{f\in FACT}\sum_{t\in PERIODS} CBUY_{rt}\cdot buy_{rft} \\
& - \sum_{p\in PRODS}\sum_{f\in FACT}\sum_{t=2}^{NT+1} CPSTOCK\cdot pstock_{pft} - \sum_{r\in RAW}\sum_{f\in FACT}\sum_{t=2}^{NT+1} CRSTOCK\cdot rstock_{rft} \\
& \forall p\in PRODS, \forall t\in PERIODS : MxSell_{pt} := \sum_{f\in FACT} sell_{pft} \leqslant MXSELL_p
\end{aligned}$$

In the decomposition algorithm, the decision variables in the main problem are expressed via the solutions (*proposals*) generated by the subproblems, such as

$$\forall p\in PRODS, \forall f\in FACT, \forall t\in PERIODS : sell_{pft} = \sum_{k=1}^{nPROP_f} Prop\_sell_{pftk}\cdot weight_{fk}$$

where $\mathrm{Prop\_sell}_{\mathrm{pftk}}$ stands for the solution value of variable $sell_{\mathrm{pt}}$ in the k^{th} proposal generated by subproblem f and $\mathrm{Prop\_cost}_{\mathrm{fk}}$ is the objective value this proposal. For every subproblem f we need to add a convexity constraint $Convex_{\mathrm{f}}$ on the $weight_{\mathrm{fk}}$ variables.

$$\text{maximize} \sum_{\mathrm{f}\in\mathrm{FACT}} \sum_{\mathrm{k}=1}^{\mathrm{nPROP_f}} \mathrm{Prop\_cost_{fk}} \cdot weight_{\mathrm{fk}}$$

$$\forall \mathrm{p} \in \mathrm{PRODS}, \forall \mathrm{t} \in \mathrm{PERIODS} : \mathrm{MxSell_{pt}} := \sum_{\mathrm{f}\in\mathrm{FACT}} \sum_{\mathrm{k}=1}^{\mathrm{nPROP_f}} \mathrm{Prop\_sell_{pftk}} \cdot weight_{\mathrm{fk}} \leqslant \mathrm{MXSELL_p}$$

$$\forall \mathrm{f} \in \mathrm{FACT} : Convex_{\mathrm{f}} := \sum_{\mathrm{k}=1}^{\mathrm{nPROP_f}} weight_{\mathrm{fk}} = 1$$

$$\forall \mathrm{f} \in \mathrm{FACT}, \forall \mathrm{k} \in 1, ..., \mathrm{nPROP_f} : weight_{\mathrm{fk}} \geqslant 0$$

We shall refer to this problem as the *modified main problem*. Without going any further into technical detail we simply remark that a correspondence exists between the solution of the original problem and those of the modified main problem.

5.2 Implementation

The decomposition algorithm has several phases:

- **Phase 1:** generation of a set of proposals to obtain a feasible solution to the modified main problem.
- **Phase 2:** optimization of the modified main problem.
- **Phase 3:** calculate the solution to the original problem.

The suproblems solved in phases 1 and 2 take as their objective functions sums of the dual values from the modfied main problem. To avoid starting off with an empty main problem and hence random dual information that may be misleading we add a crash *Phase 0* to our implementation that generates one proposal for each subproblem with the original objective function.

5.2.1 Main model

Below follows the body of the main model file. The definitions of the function bodies will be shown later in Section 5.2.3.

```
model "Coco3 Main"
 uses "mmxprs","mmjobs","mmsystem"

 parameters
  DATAFILE = "coco2.dat"
  ALG = 0                                ! 0: stop phase with 1st failed subpb.
                                         ! 1: stop when all subprob.s fail
 end-parameters

 forward procedure process_sub_result
 forward procedure solve_main(phase:integer)
 forward procedure process_main_result
 forward function calc_solution:real
 forward procedure print_solution
```

```
 declarations
  PHASE_0=2                          ! Event codes sent to submodels
  PHASE_1=3
  PHASE_2=4
  PHASE_3=5
  EVENT_SOLVED=6                     ! Event codes sent by submodels
  EVENT_FAILED=7
  EVENT_READY=8
  NPROD, NFACT, NRAW, NT: integer
 end-declarations

 initializations from DATAFILE
  NPROD NFACT NRAW NT
 end-initializations

 declarations
  PRODS = 1..NPROD                   ! Range of products (p)
  FACT = 1..NFACT                    !          factories (f)
  RAW = 1..NRAW                      !          raw materials (r)
  PERIODS = 1..NT                    !          time periods (t)

  nIter: integer                     ! Iteration counter
  nPROP: array(FACT) of integer      ! Counters of proposals from subprob.s
 end-declarations

!**** Main problem ****
 declarations
  MXSELL: array(PRODS,PERIODS) of real   ! Max. amount of p that can be sold
  excessS: mpvar                         ! Violation of sales/buying limits
  weight: array(FACT,range) of mpvar     ! weights for proposals
  MxSell: array(PRODS,PERIODS) of linctr ! Sales limit constraints
  Convex: array(FACT) of linctr          ! Convexity constraints
  Price_convex: array(FACT) of real      ! Dual price on convexity constraints
  Price_sell: array(PRODS,PERIODS) of real ! Dual price on sales limits
 end-declarations

 initializations from DATAFILE
  MXSELL
 end-initializations

!**** Submodels ****
 declarations
  submod: array(FACT) of Model       ! One subproblem per factory
  Stopped: set of integer
 end-declarations

 res:= compile("g","cocoSubF.mos")   ! Compile the submodel file
 forall(f in FACT) do                ! Load & run one submodel per product
  Price_convex(f):= 1
  load(submod(f), "cocoSubF.bim")
  submod(f).uid:= f
  run(submod(f), "Factory=" + f + ",DATAFILE=" + DATAFILE)
  wait                               ! Wait for child model to be ready
  ev:=getnextevent
  if ev.class=EVENT_END then
   writeln_("Cannot start all necessary models - aborting")
   exit(1)
  end-if
 end-do

!**** Phase 0: Crash ****
 nIter:=1; finished:=false
 writeln_("\nPHASE 0 -- Iteration ", nIter); fflush

 forall(f in FACT)                   ! Start solving all submodels (Phase 1)
  send(submod(f), PHASE_0, 0)
```

```
 forall(f in FACT) do
  wait                                ! Wait for child (termination) events
  ev:= getnextevent
  if getclass(ev)=EVENT_SOLVED then
   process_sub_result                 ! Add new proposal to main problem
  elif getclass(ev)=EVENT_FAILED then
   finished:= true
  end-if
 end-do

 if finished then
  writeln_("Problem is infeasible")
  exit(1)
 end-if

 solve_main(1)                        ! Solve the updated Ph. 1 main problem
 process_main_result                  ! Store initial pricing data for submodels


!**** Phase 1: proposal generation (feasibility) ****
 repeat
  noimprove:= 0
  nIter+=1
  writeln_("\nPHASE 1 -- Iteration ", nIter); fflush

  forall(f in FACT)                   ! Start solving all submodels (Phase 1)
   send(submod(f), PHASE_1, Price_convex(f))

  forall(f in FACT) do
   wait                               ! Wait for child (termination) events
   ev:= getnextevent
   if getclass(ev)=EVENT_SOLVED then
    process_sub_result                ! Add new proposal to main problem
   elif getclass(ev)=EVENT_FAILED then
    noimprove += 1
   end-if
  end-do

  if noimprove = NFACT then
   writeln_("Problem is infeasible")
   exit(2)
  end-if
  if ALG=0 and noimprove > 0 then
   writeln_("No improvement by some subproblem(s)")
   break
  end-if

  solve_main(1)                       ! Solve the updated Ph. 1 main problem
  if getobjval>0.00001 then
   process_main_result                ! Store new pricing data for submodels
  end-if
 until getobjval <= 0.00001


!**** Phase 2: proposal generation (optimization) ****
 writeln_("\n**** PHASE 2 ****")
 finished:=false
 repeat
  solve_main(2)                       ! Solve Phase 2 main problem
  process_main_result                 ! Store new pricing data for submodels

  nIter+=1
  writeln_("\nPHASE 2 -- Iteration ", nIter); fflush

  forall(f in FACT)                   ! Start solving all submodels (Phase 2)
   send(submod(f), PHASE_2, Price_convex(f))
```

```
  forall(f in FACT) do
   wait                                ! Wait for child (termination) events
   ev:= getnextevent
   if getclass(ev)=EVENT_SOLVED then
    process_sub_result                 ! Add new proposal to main problem
   elif getclass(ev)=EVENT_FAILED then
    if ALG=0 then
     finished:=true                    ! 1st submodel w/o prop. stops phase 2
    else
     Stopped += {ev.fromuid}           ! Stop phase 2 only when no submodel
                                       ! generates a new proposal
    end-if
   end-if
  end-do

  if getsize(Stopped) = NFACT then finished:= true; end-if
 until finished

 solve_main(2)                         ! Re-solve main to integrate
                                       ! proposal(s) from last ph. 2 iteration

!**** Phase 3: solution to the original problem ****
 writeln_("\n**** PHASE 3 ****")
 forall(f in FACT) do
  send(submod(f), PHASE_3, 0)          ! Stop all submodels
  wait
  dropnextevent
 end-do

 writeln_("Total Profit=", calc_solution)
 print_solution
end-model
```

The initial `declarations` block of this model defines a certain number of event codes that are used to identify the messages sent between the controlling (main) model and its child (sub)models. The same declarations need to be repeated in the child models.

5.2.2 Single factory model

The model file `cocoSubF.mos` with the definition of the subproblems has the following contents.

```
model "Coco Subproblem (factory based decomp.)"
 uses "mmxprs", "mmjobs"

 parameters
  Factory = 0
  TOL = 0.00001
  DATAFILE = "coco3.dat"
 end-parameters

 forward procedure process_solution

 declarations
  PHASE_0=2                            ! Event codes sent to submodels
  PHASE_1=3
  PHASE_2=4
  PHASE_3=5
  EVENT_SOLVED=6                       ! Event codes sent by submodels
  EVENT_FAILED=7
  EVENT_READY=8
  NPROD, NFACT, NRAW, NT: integer
 end-declarations

 send(EVENT_READY,0)                   ! Model is ready (= running)
```

```
 initializations from DATAFILE
  NPROD NFACT NRAW NT
 end-initializations

 declarations
  PRODS = 1..NPROD                      ! Range of products (p)
  FACT = 1..NFACT                       !          factories (f)
  RAW = 1..NRAW                         !          raw materials (r)
  PERIODS = 1..NT                       !          time periods (t)

  REV: array(PRODS,PERIODS) of real     ! Unit selling price of products
  CMAKE: array(PRODS,FACT) of real      ! Unit cost to make product p
                                        ! at factory f
  CBUY: array(RAW,PERIODS) of real      ! Unit cost to buy raw materials
  REQ: array(PRODS,RAW) of real         ! Requirement by unit of product p
                                        ! for raw material r
  MXSELL: array(PRODS,PERIODS) of real  ! Max. amount of p that can be sold
  MXMAKE: array(FACT) of real           ! Max. amount factory f can make
                                        ! over all products
  IPSTOCK: array(PRODS,FACT) of real    ! Initial product stock levels
  IRSTOCK: array(RAW,FACT) of real      ! Initial raw material stock levels
  CPSTOCK: real                         ! Unit cost to store any product p
  CRSTOCK: real                         ! Unit cost to store any raw mat. r
  MXRSTOCK: real                        ! Raw material storage capacity

  make: array(PRODS,PERIODS) of mpvar   ! Amount of products made at factory
  sell: array(PRODS,PERIODS) of mpvar   ! Amount of product sold from factory
  buy: array(RAW,PERIODS) of mpvar      ! Amount of raw material bought
  pstock: array(PRODS,1..NT+1) of mpvar ! Product stock levels at start
                                        ! of period t
  rstock: array(RAW,1..NT+1) of mpvar   ! Raw material stock levels
                                        ! at start of period t

  sol_make: array(PRODS,PERIODS) of real   ! Amount of products made
  sol_sell: array(PRODS,PERIODS) of real   ! Amount of product sold
  sol_buy: array(RAW,PERIODS) of real      ! Amount of raw mat. bought
  sol_pstock: array(PRODS,1..NT+1) of real ! Product stock levels
  sol_rstock: array(RAW,1..NT+1) of real   ! Raw mat. stock levels

  Profit: linctr                        ! Profit of proposal
  Price_sell: array(PRODS,PERIODS) of real ! Dual price on sales limits
 end-declarations

 initializations from DATAFILE
  CMAKE REV CBUY REQ MXSELL MXMAKE
  IPSTOCK IRSTOCK MXRSTOCK CPSTOCK CRSTOCK
 end-initializations

! Product stock balance
 forall(p in PRODS,t in PERIODS)
  PBal(p,t):= pstock(p,t+1) = pstock(p,t) + make(p,t) - sell(p,t)

! Raw material stock balance
 forall(r in RAW,t in PERIODS)
  RBal(r,t):= rstock(r,t+1) =
   rstock(r,t) + buy(r,t) - sum(p in PRODS) REQ(p,r)*make(p,t)

! Capacity limit
 forall(t in PERIODS)
  MxMake(t):= sum(p in PRODS) make(p,t) <= MXMAKE(Factory)

! Limit on the amount of prod. p to be sold
 forall(p in PRODS,t in PERIODS) sell(p,t) <= MXSELL(p,t)

! Raw material stock limit
 forall(t in 2..NT+1)
  MxRStock(t):= sum(r in RAW) rstock(r,t) <= MXRSTOCK
```

```
! Initial product and raw material stock levels
 forall(p in PRODS) pstock(p,1) = IPSTOCK(p,Factory)
 forall(r in RAW) rstock(r,1) = IRSTOCK(r,Factory)

! Total profit
 Profit:=
  sum(p in PRODS,t in PERIODS) REV(p,t) * sell(p,t) -         ! revenue
  sum(p in PRODS,t in PERIODS) CMAKE(p,Factory) * make(p,t) - ! prod. cost
  sum(r in RAW,t in PERIODS) CBUY(r,t) * buy(r,t) -           ! raw mat.
  sum(p in PRODS,t in 2..NT+1) CPSTOCK * pstock(p,t) -        ! p storage
  sum(r in RAW,t in 2..NT+1) CRSTOCK * rstock(r,t)            ! r storage

! (Re)solve this model until it is stopped by event "PHASE_3"
 repeat
  wait
  ev:= getnextevent
  Phase:= getclass(ev)
  if Phase=PHASE_3 then               ! Stop the execution of this model
   break
  end-if
  Price_convex:= getvalue(ev)         ! Get new pricing data

  if Phase<>PHASE_0 then
   initializations from "bin:shmem:pricedata"
    Price_sell
   end-initializations
  end-if

 ! (Re)solve this model
  if Phase=PHASE_0 then
   maximize(Profit)
  elif Phase=PHASE_1 then
   maximize(sum(p in PRODS,t in PERIODS) Price_sell(p,t)*sell(p,t) + Price_convex)
  else        ! PHASE 2
   maximize(
    Profit - sum(p in PRODS,t in PERIODS) Price_sell(p,t)*sell(p,t) -
     Price_convex)
  end-if

  writeln_("Factory ", Factory, " - Obj: ", getobjval,
           " Profit: ", getsol(Profit), " Price_sell: ",
           getsol(sum(p in PRODS,t in PERIODS) Price_sell(p,t)*sell(p,t) ),
           " Price_convex: ", Price_convex)
  fflush

  if getobjval > TOL then             ! Solution found: send values to parent
   process_solution
  elif getobjval <= TOL then          ! Problem is infeasible (Phase 0/1) or
   send(EVENT_FAILED,0)               ! no improved solution found (Phase 2)
  else
   send(EVENT_READY,0)
  end-if
 until false

!-------------------------------------------------------------
! Process solution data
 procedure process_solution
  forall(p in PRODS,t in PERIODS) do
   sol_make(p,t):= getsol(make(p,t))
   sol_sell(p,t):= getsol(sell(p,t))
  end-do
  forall(r in RAW,t in PERIODS) sol_buy(r,t):= getsol(buy(r,t))
  forall(p in PRODS,t in 1..NT+1) sol_pstock(p,t):= getsol(pstock(p,t))
  forall(r in RAW,t in 1..NT+1) sol_rstock(r,t):= getsol(rstock(r,t))
  Prop_cost:= getsol(Profit)
  send(EVENT_SOLVED,0)
```

```
  initializations to "mempipe:noindex,sol"
   Factory
   sol_make sol_sell sol_buy sol_pstock sol_rstock
   Prop_cost
  end-initializations
 end-procedure
end-model
```

The child models are re-solved until they receive the `PHASE_3` code. At every iteration they write their solution values to memory so that these can be processed by the parent model.

5.2.3 Main model subroutines

The following three subroutines are defined in the main model to recover the solutions produced by the subproblems (`process_sub_result`), re-solve the main problem (`solve_main`), and communicate the solution of the main problem to the child models (`process_main_result`).

```
 declarations
  Prop_make: array(PRODS,FACT,PERIODS,range) of real ! Amount of products made
  Prop_sell: array(PRODS,FACT,PERIODS,range) of real ! Amount of product sold
  Prop_buy: array(RAW,FACT,PERIODS,range) of real    ! Amount of raw mat. bought
  Prop_pstock: array(PRODS,FACT,1..NT+1,range) of real ! Product stock levels
  Prop_rstock: array(RAW,FACT,1..NT+1,range) of real   ! Raw mat. stock levels
  Prop_cost: array(FACT,range) of real  ! Cost/profit of each proposal
 end-declarations

 procedure process_sub_result
  declarations
   f: integer                          ! Factory index
                                       ! Solution values of the proposal:
   sol_make: array(PRODS,PERIODS) of real      ! Amount of products made
   sol_sell: array(PRODS,PERIODS) of real      ! Amount of product sold
   sol_buy: array(RAW,PERIODS) of real         ! Amount of raw mat. bought
   sol_pstock: array(PRODS,1..NT+1) of real ! Product stock levels
   sol_rstock: array(RAW,1..NT+1) of real   ! Raw mat. stock levels
   pc: real                            ! Cost of the proposal
  end-declarations

 ! Read proposal data from memory
  initializations from "mempipe:noindex,sol"
   f as "Factory"
   sol_make sol_sell sol_buy sol_pstock sol_rstock
   pc as "Prop_cost"
  end-initializations

 ! Add the new proposal to the main problem
  nPROP(f)+=1
  create(weight(f,nPROP(f)))
  forall(p in PRODS,t in PERIODS) do
   Prop_make(p,f,t,nPROP(f)):= sol_make(p,t)
   Prop_sell(p,f,t,nPROP(f)):= sol_sell(p,t)
  end-do
  forall(r in RAW,t in PERIODS) Prop_buy(r,f,t,nPROP(f)):= sol_buy(r,t)
  forall(p in PRODS,t in 1..NT+1) Prop_pstock(p,f,t,nPROP(f)):= sol_pstock(p,t)
  forall(r in RAW,t in 1..NT+1) Prop_rstock(r,f,t,nPROP(f)):= sol_rstock(r,t)
  Prop_cost(f,nPROP(f)):= pc
  writeln_("Sol. for factory ", f, ":\n  make:   ", sol_make, "\n  sell:   ",
           sol_sell, "\n  buy:    ", sol_buy, "\n  pstock: ", sol_pstock,
    "\n  rstock: ", sol_rstock)
 end-procedure

!-------------------------------------------------------------
 procedure solve_main(phase: integer)
  forall(f in FACT)
```

```
   Convex(f):= sum (k in 1..nPROP(f)) weight(f,k) = 1

  if phase=1 then
   forall(p in PRODS,t in PERIODS)
    MxSell(p,t):=
     sum(f in FACT,k in 1..nPROP(f)) Prop_sell(p,f,t,k)*weight(f,k) -
      excessS <= MXSELL(p,t)
   minimize(excessS)
  else
   forall(p in PRODS,t in PERIODS)
    MxSell(p,t):=
     sum(f in FACT,k in 1..nPROP(f)) Prop_sell(p,f,t,k)*weight(f,k) <=
      MXSELL(p,t)
   maximize(sum(f in FACT, k in 1..nPROP(f)) Prop_cost(f,k) * weight(f,k))
  end-if

  writeln_("Main problem objective: ", getobjval)
  write_("  Weights:")
  forall(f in FACT,k in 1..nPROP(f)) write(" ", getsol(weight(f,k)))
  writeln
 end-procedure

!-------------------------------------------------------------
 procedure process_main_result
  forall(p in PRODS,t in PERIODS) Price_sell(p,t):=getdual(MxSell(p,t))
  forall(f in FACT) Price_convex(f):=getdual(Convex(f))

  initializations to "bin:shmem:pricedata"
   Price_sell
  end-initializations
 end-procedure
```

Finally, the main model is completed by two subroutines for calculating the solution to the original problem (`calc_solution`), Phase 3 of the decomposition algorithm, and printing out the solution (`print_solution`). The solution to the original problem is obtained from the solution values of the modfied main problem and the proposals generated by the subproblems.

```
 declarations
  REV: array(PRODS,PERIODS) of real   ! Unit selling price of products
  CMAKE: array(PRODS,FACT) of real    ! Unit cost to make product p
                                      ! at factory f
  CBUY: array(RAW,PERIODS) of real    ! Unit cost to buy raw materials
  COPEN: array(FACT) of real          ! Fixed cost of factory f being
                                      ! open for one period
  CPSTOCK: real                       ! Unit cost to store any product p
  CRSTOCK: real                       ! Unit cost to store any raw mat. r

  Sol_make: array(PRODS,FACT,PERIODS) of real ! Solution value (products made)
  Sol_sell: array(PRODS,FACT,PERIODS) of real ! Solution value (product sold)
  Sol_buy: array(RAW,FACT,PERIODS) of real    ! Solution value (raw mat. bought)
  Sol_pstock: array(PRODS,FACT,1..NT+1) of real ! Sol. value (prod. stock)
  Sol_rstock: array(RAW,FACT,1..NT+1) of real   ! Sol. value (raw mat. stock)
 end-declarations

 initializations from DATAFILE
  CMAKE REV CBUY CPSTOCK CRSTOCK COPEN
 end-initializations

 function calc_solution: real
  forall(p in PRODS,f in FACT,t in PERIODS) do
   Sol_sell(p,f,t):=
    sum(k in 1..nPROP(f)) Prop_sell(p,f,t,k) * getsol(weight(f,k))
   Sol_make(p,f,t):=
    sum(k in 1..nPROP(f)) Prop_make(p,f,t,k) * getsol(weight(f,k))
  end-do
```

```
  forall(r in RAW,f in FACT,t in PERIODS) Sol_buy(r,f,t):=
    sum(k in 1..nPROP(f)) Prop_buy(r,f,t,k) * getsol(weight(f,k))
  forall(p in PRODS,f in FACT,t in 1..NT+1) Sol_pstock(p,f,t):=
   sum(k in 1..nPROP(f)) Prop_pstock(p,f,t,k) * getsol(weight(f,k))
  forall(r in RAW,f in FACT,t in 1..NT+1) Sol_rstock(r,f,t):=
   sum(k in 1..nPROP(f)) Prop_rstock(r,f,t,k) * getsol(weight(f,k))

  returned:=
   sum(p in PRODS,f in FACT,t in PERIODS) REV(p,t) * Sol_sell(p,f,t) -
   sum(p in PRODS,f in FACT,t in PERIODS) CMAKE(p,f) * Sol_make(p,f,t) -
   sum(r in RAW,f in FACT,t in PERIODS) CBUY(r,t) * Sol_buy(r,f,t) -
   sum(p in PRODS,f in FACT,t in 2..NT+1) CPSTOCK * Sol_pstock(p,f,t) -
   sum(r in RAW,f in FACT,t in 2..NT+1) CRSTOCK * Sol_rstock(r,f,t)
 end-function

!-------------------------------------------------------------
 procedure print_solution
  writeln_("Finished products:")
  forall(f in FACT) do
   writeln_("Factory ", f, ":")
   forall(p in PRODS) do
    write("  ", p, ":    ")
    forall(t in PERIODS) write(strfmt(Sol_make(p,f,t),6,1), "(",
                               strfmt(Sol_pstock(p,f,t+1),5,1), ")")
    writeln
   end-do
  end-do

  writeln_("Raw material:")
  forall(f in FACT) do
   writeln_("Factory ", f, ":")
   forall(r in RAW) do
    write("  ", r, ": ")
    forall(t in PERIODS) write(strfmt(Sol_buy(r,f,t),6,1), "(",
                               strfmt(Sol_rstock(r,f,t+1),5,1), ")")
    writeln
   end-do
  end-do

  writeln_("Sales:")
  forall(f in FACT) do
   writeln_("Factory ", f, ":")
   forall(p in PRODS) do
    write("  ", p, ": ")
    forall(t in PERIODS) write(strfmt(Sol_sell(p,f,t),4))
    writeln
   end-do
  end-do

  writeln_("\nComputation time: ", gettime)
 end-procedure
```

5.3 Results

For the test cases that have been tried the solutions produced by our decomposition algorithm are close to the optimal solution, but the latter is not always reached. The reason behind this is that the decomposition algorithm is a sequence of iterations that may accumulate errors at different levels—lowering the tolerances used as stopping criterion in the submodels most of the time does not improve the solution. However, the configuration of the decomposition algorithm itself shows some impact on the solution: in phases 1 and 2 one may choose, for instance, to stop once the first submodel returns no improvement or continue until no more proposals are generated. Generating more proposals sometimes helps finding a better solution, but it also increases the number of times (sub)problems are solved and hence prolongates the solving time.

6 Benders decomposition: working with several different submodels

Benders decomposition is a decomposition method for solving large Mixed Integer Programming problems. Instead of solving a MIP problem that may be too large for standard solution methods all-in-one, we work with a sequence of linear and pure integer subproblems (the latter with a smaller number of constraints than the original problem). The description of the decomposition algorithm below is taken from [**?**] where the interested reader will also find proofs of its finiteness and of the statement that it always results in the optimal solution.

Consider the following standard form of a mixed-integer programming problem (*problem I*).

$$\begin{aligned} &\text{minimize } z = \mathbf{Cx} + \mathbf{Dy} \\ &\mathbf{Ax} + \mathbf{By} \geqslant \mathbf{b} \\ &\mathbf{x}, \mathbf{y} \geqslant \mathbf{0}, \mathbf{y} \text{ integer} \end{aligned}$$

In the above, and all through this section, we use bold letters for vectors and matrices. For instance, $\mathbf{Cx} + \mathbf{Dy}$ stands for $\sum_{i=1}^{NCTVAR} C_i \cdot x_i + \sum_{i=1}^{NINTVAR} D_i \cdot y_i$, where $NCTVAR$ denotes the number of continuous variables and $NINTVAR$ the number of integer variables.

For given values $\mathbf{y}'$ of $\mathbf{y}$ the problem above is reduced to a linear program (*problem II*)—we leave out the constant term in the objective:

$$\begin{aligned} &\text{minimize } \mathbf{Cx} \\ &\mathbf{Ax} \geqslant \mathbf{b} - \mathbf{By}' \\ &\mathbf{x} \geqslant \mathbf{0} \end{aligned}$$

The dual program of *problem II* is given by *problem IId*:

$$\begin{aligned} &\text{maximize } \mathbf{u}(\mathbf{b} - \mathbf{By}') \\ &\mathbf{uA} \leqslant \mathbf{C} \\ &\mathbf{u} \geqslant \mathbf{0} \end{aligned}$$

An interesting feature of the dual problem *IId* is that the feasible region of $\mathbf{u}$ is independent of $\mathbf{y}$. Furthermore, from duality theory it follows that if problem IId is infeasible or has no finite optimum solution, then the original problem I has no finite optimum solution. Again from duality theory we know that if problem IId has a finite optimum solution $\mathbf{u}^*$ then this solution has the same value as the optimum solution to the primal problem (that is, $\mathbf{u}^*(\mathbf{b} - \mathbf{By}') = \mathbf{Cx}^*$), and for a solution $\mathbf{u}^p$ (at a vertex p of the feasible region) we have $\mathbf{u}^p(\mathbf{b} - \mathbf{By}') \leqslant \mathbf{Cx}^*$. Substituting the latter into the objective of the original MIP problem we obtain a constraint of the form

$$z \geqslant \mathbf{u}^p(\mathbf{b} - \mathbf{By}) + \mathbf{Dy}$$

To obtain the optimum solution $\mathbf{z}^*$ of the original MIP problem (problem I) we may use the following partitioning algorithm that is known as *Benders decomposition algorithm*:

Step 0 Find a $\mathbf{u}'$ that satisfies $\mathbf{u'A} \leqslant \mathbf{C}$
if no such $\mathbf{u}'$ exists
then the original problem (I) has no feasible solution
else continue with Step 1
end-if

Step 1 Solve the pure integer program

$$\begin{aligned}&\text{minimize } z\\ &z \geqslant \mathbf{u}'(\mathbf{b} - \mathbf{B}\mathbf{y}) + \mathbf{D}\mathbf{y}\\ &\mathbf{y} \geqslant \mathbf{0}, \mathbf{y} \text{ integer}\end{aligned}$$

If z is unbounded from below, take a z to be any small value z'.

Step 2 With the solution $\mathbf{y}'$ of Step 1, solve the linear program

$$\begin{aligned}&\text{maximize } \mathbf{u}(\mathbf{b} - \mathbf{B}\mathbf{y}')\\ &\mathbf{u}\mathbf{A} \leqslant \mathbf{C}\\ &\mathbf{u} \geqslant \mathbf{0}\end{aligned}$$

If $\mathbf{u}$ goes to inifinity with $\mathbf{u}(\mathbf{b} - \mathbf{B}\mathbf{y}')$
then add the constraint $\sum_i u_i \leqslant M$, where M is a large positive constant, and resolve the problem
end-if
Let the solution of this program be $\mathbf{u}''$.
if $z' - \mathbf{D}\mathbf{y}' \geqslant \mathbf{u}''(\mathbf{b} - \mathbf{B}\mathbf{y}')$
then continue with Step 3
else return to Step 1 and add the constraint $z' \geqslant \mathbf{D}\mathbf{y} + \mathbf{u}''(\mathbf{b} - \mathbf{B}\mathbf{y})$
end-if

Step 3 With the solution $\mathbf{y}'$ of Step 1, solve the linear program

$$\begin{aligned}&\text{minimize } \mathbf{C}\mathbf{x}\\ &\mathbf{A}\mathbf{x} \geqslant \mathbf{b} - \mathbf{B}\mathbf{y}'\\ &\mathbf{x} \geqslant \mathbf{0}\end{aligned}$$

$\mathbf{x}'$ and $\mathbf{y}'$ are the optimum solution $z^* = \mathbf{C}\mathbf{x}' + \mathbf{D}\mathbf{y}'$

This algorithm is provably finite. It results in the optimum solution and at any time during its execution lower and upper bounds on the optimum solution $\mathbf{z}^*$ can be obtained.

6.1 A small example problem

Our implementation of Benders decomposition will solve the following example problem with $\mathrm{NCTVAR} = 3$ continuous variables x_i, $\mathrm{NINTVAR} = 3$ integer variables y_i, and $\mathrm{NC} = 4$ inequality constraints.

$$\begin{aligned}\text{minimize } & 5\cdot x_1 + 4\cdot x_2 + 3\cdot x_3 + 2\cdot y_1 + 2\cdot y_2 + 3\cdot y_3 \\ & x_1 - x_2 + 5\cdot x_3 + 3\cdot y_1 + 2\cdot y_3 \geqslant 5\\ & 4\cdot x_2 + 3\cdot x_3 - y_1 + y_2 \geqslant 7\\ & 2\cdot x_1 - 2\cdot x_3 + y_1 - y_2 \geqslant 4\\ & 3\cdot x_1 + 5\cdot y_1 + 5\cdot y_2 + 5\cdot y_3 \geqslant -2\\ & \mathbf{x}, \mathbf{y} \geqslant \mathbf{0},\ \mathbf{y} \text{ integer}\end{aligned}$$

6.2 Implementation

6.2.1 Main model

The main model reads in the data, defines the solution algorithm, coordinates the communication between the submodels, and prints out the solution at the end. For step 2 of the algorithm (solving the dual problem with fixed integer variables) we have the choice to solve either the primal problem and retrieve the dual solution values from the Optimizer or to define the dual problem ourselves and solve it. Model parameter `ALG` lets the user choose between these two options.

The implementation of the main model looks as follows. Prior to the start of the solution algorithm itself all submodels are compiled, loaded, and started so that in each step of the algorithm we simply need to trigger the (re)solving of the corresponding submodel.

```
model "Benders (main model)"
 uses "mmxprs", "mmjobs"

 parameters
  NCTVAR = 3
  NINTVAR = 3
  NC = 4
  BIGM = 1000
  ALG = 1                              ! 1: Use Benders decomposition (dual)
                                       ! 2: Use Benders decomposition (primal)
  DATAFILE = "bprob33.dat"
 end-parameters

 forward procedure start_solution
 forward procedure solve_primal_int(ct: integer)
 forward procedure solve_cont
 forward function eval_solution: boolean
 forward procedure print_solution

 declarations
  STEP_0=2                             ! Event codes sent to submodels
  STEP_1=3
  STEP_2=4
  EVENT_SOLVED=6                       ! Event codes sent by submodels
  EVENT_INFEAS=7
  EVENT_READY=8

  CtVars = 1..NCTVAR                   ! Continuous variables
  IntVars = 1..NINTVAR                 ! Discrete variables
  Ctrs = 1..NC                         ! Set of constraints (orig. problem)

  A: array(Ctrs,CtVars) of integer     ! Coeff.s of continuous variables
  B: array(Ctrs,IntVars) of integer    ! Coeff.s of discrete variables
  b: array(Ctrs) of integer            ! RHS values
  C: array(CtVars) of integer          ! Obj. coeff.s of continuous variables
  D: array(IntVars) of integer         ! Obj. coeff.s of discrete variables
  Ctr: array(Ctrs) of linctr           ! Constraints of orig. problem
  CtrD: array(CtVars) of linctr        ! Constraints of dual problem
  MC: array(range) of linctr           ! Constraints generated by alg.

  sol_u: array(Ctrs) of real           ! Solution of dual problem
  sol_x: array(CtVars) of real         ! Solution of primal prob. (cont.)
  sol_y: array(IntVars) of real        ! Solution of primal prob. (integers)
  sol_obj: real                        ! Objective function value (primal)

  RM: range                            ! Model indices
  stepmod: array(RM) of Model          ! Submodels
 end-declarations

 initializations from DATAFILE
```

```
  A B b C D
 end-initializations

! **** Submodels ****

 initializations to "bin:shmem:probdata"   ! Save data for submodels
  A  B  b  C  D
 end-initializations

! Compile + load all submodels
 if compile("benders_int.mos")<>0: exit(1)
 create(stepmod(1)); load(stepmod(1), "benders_int.bim")
 if compile("benders_dual.mos")<>0: exit(2)

 if ALG=1 then
  create(stepmod(2)); load(stepmod(2), "benders_dual.bim")
 else
  create(stepmod(0)); load(stepmod(0), "benders_dual.bim")
  if compile("benders_cont.mos")<>0: exit(3)
  create(stepmod(2)); load(stepmod(2), "benders_cont.bim")
  run(stepmod(0), "NCTVAR=" + NCTVAR + ",NINTVAR=" + NINTVAR + ",NC=" + NC)
 end-if
                                    ! Start the execution of the submodels
 run(stepmod(1), "NINTVAR=" + NINTVAR + ",NC=" + NC)
 run(stepmod(2), "NCTVAR=" + NCTVAR + ",NINTVAR=" + NINTVAR + ",NC=" + NC)

 forall(m in RM) do
  wait                              ! Wait for "Ready" messages
  ev:= getnextevent
  if getclass(ev) <> EVENT_READY then
   writeln_("Error occurred in a subproblem")
   exit(4)
  end-if
 end-do

! **** Solution algorithm ****

 start_solution                     ! 0. Initial solution for cont. var.s
 ct:= 1
 repeat
 writeln_("\n**** Iteration: ", ct)
  solve_primal_int(ct)              ! 1. Solve problem with fixed cont.
  solve_cont                        ! 2. Solve problem with fixed int.
  ct+=1
 until eval_solution                !    Test for optimality
 print_solution                     ! 3. Retrieve and display the solution
```

The subroutines starting the different submodels send a 'start solving' event and retrieve the solution once the submodel solving has finished. For the generation of the start solution we need to choose the right submodel, according to the settings of the parameter `ALG`. If this problem is found to be infeasible, then the whole problem is infeasible and we stop the execution of the model.

```
! Produce an initial solution for the dual problem using a dummy objective
 procedure start_solution
  if ALG=1 then                     ! Start the problem solving
   send(stepmod(2), STEP_0, 0)
  else
   send(stepmod(0), STEP_0, 0)
  end-if
  wait                              ! Wait for the solution
  ev:=getnextevent
  if getclass(ev)=EVENT_INFEAS then
   writeln_("Problem is infeasible")
   exit(6)
```

```
  end-if
 end-procedure

!-------------------------------------------------------------
! Solve a modified version of the primal problem, replacing continuous
! variables by the solution of the dual
 procedure solve_primal_int(ct: integer)
  send(stepmod(1), STEP_1, ct)        ! Start the problem solving
  wait                                ! Wait for the solution
  ev:=getnextevent
  sol_obj:= getvalue(ev)              ! Store objective function value

  initializations from "bin:shmem:sol"  ! Retrieve the solution
   sol_y
  end-initializations
 end-procedure

!-------------------------------------------------------------
! Solve the Step 2 problem (dual or primal depending on value of ALG)
! for given solution values of y
 procedure solve_cont
  send(stepmod(2), STEP_2, 0)         ! Start the problem solving
  wait                                ! Wait for the solution
  dropnextevent

  initializations from "bin:shmem:sol"  ! Retrieve the solution
   sol_u
  end-initializations
 end-procedure
```

The main model also tests whether the termination criterion is fulfilled (function `eval_solution`) and prints out the final solution (procedure `print_solution`). The latter procedure also stops all submodels:

```
 function eval_solution: boolean
  write_("Test optimality: ", sol_obj - sum(i in IntVars) D(i)*sol_y(i),
         " >= ", sum(j in Ctrs) sol_u(j)* (b(j) -
                           sum(i in IntVars) B(j,i)*sol_y(i)) )
  returned:= ( sol_obj - sum(i in IntVars) D(i)*sol_y(i) >=
      sum(j in Ctrs) sol_u(j)* (b(j) - sum(i in IntVars) B(j,i)*sol_y(i)) )
  writeln_(if(returned, " : true", " : false"))
 end-function

!-------------------------------------------------------------
 procedure print_solution
  ! Retrieve results
  initializations from "bin:shmem:sol"
   sol_x
  end-initializations

  forall(m in RM) stop(stepmod(m))     ! Stop all submodels

  write_("\n**** Solution (Benders): ", sol_obj, "\n  x: ")
  forall(i in CtVars) write(sol_x(i), " ")
  write("  y: ")
  forall(i in IntVars) write(sol_y(i), " ")
  writeln
 end-procedure
```

6.2.2 Submodel 1: fixed continuous variables

In the first step of the decomposition algorithm we need to solve a pure integer problem. When the execution of this model is started it reads in the invariant data and sets up the variables. It then halts at the `wait` statement (first line of the `repeat-until` loop) until the parent model sends it a (solving)

event. At each invocation of solving this problem, the solution values of the previous run of the continuous problem—read from memory—are used to define a new constraint `MC(k)` for the integer problem. The whole model, and with it the endless loop into which the solving is embedded, will be terminated only by the 'stop model' command from the parent model. The complete source of this submodel (file `benders_int.mos`) is listed below.

```
model "Benders (integer problem)"
 uses "mmxprs", "mmjobs"

 parameters
  NINTVAR = 3
  NC = 4
  BIGM = 1000
 end-parameters

 declarations
  STEP_0=2                            ! Event codes sent to submodels
  STEP_1=3
  EVENT_SOLVED=6                      ! Event codes sent by submodels
  EVENT_READY=8

  IntVars = 1..NINTVAR                ! Discrete variables
  Ctrs = 1..NC                        ! Set of constraints (orig. problem)

  B: array(Ctrs,IntVars) of integer   ! Coeff.s of discrete variables
  b: array(Ctrs) of integer           ! RHS values
  D: array(IntVars) of integer        ! Obj. coeff.s of discrete variables
  MC: array(range) of linctr          ! Constraints generated by alg.

  sol_u: array(Ctrs) of real          ! Solution of dual problem
  sol_y: array(IntVars) of real       ! Solution of primal prob.

  y: array(IntVars) of mpvar          ! Discrete variables
  z: mpvar                            ! Objective function variable
 end-declarations

 initializations from "bin:shmem:probdata"
  B  b  D
 end-initializations

 z is_free                            ! Objective is a free variable
 forall(i in IntVars) y(i) is_integer ! Integrality condition
 forall(i in IntVars) y(i) <= BIGM    ! Set (large) upper bound

 send(EVENT_READY,0)                  ! Model is ready (= running)

 repeat
  wait
  ev:= getnextevent
  ct:= integer(getvalue(ev))

  initializations from "bin:shmem:sol"
   sol_u
  end-initializations

  ! Add a new constraint
  MC(ct):= z >= sum(i in IntVars) D(i)*y(i) +
               sum(j in Ctrs) sol_u(j)*(b(j) - sum(i in IntVars) B(j,i)*y(i))

  minimize(z)

  ! Store solution values of y
  forall(i in IntVars) sol_y(i):= getsol(y(i))

  initializations to "bin:shmem:sol"
   sol_y
```

```
  end-initializations

  send(EVENT_SOLVED, getobjval)

  write_("Step 1: ", getobjval, "\n  y: ")
  forall(i in IntVars) write(sol_y(i), " ")

  write_("\n  Slack: ")
  forall(j in 1..ct) write(getslack(MC(j)), " ")
  writeln
  fflush

 until false

end-model
```

Since the problems we are solving differ only by a single constraint from one iteration to the next, it may be worthwhile to save the basis of the solution to the root LP-relaxation (*not* the basis to the MIP solution) and reload it for the next optimization run. However, for our small test case we did not observe any improvements in terms of execution speed. For saving and re-reading the basis, the call to `minimize` needs to be replaced by the following sequence of statements:

```
declarations
 bas: basis
end-declarations

loadprob(z)
loadbasis(bas)
minimize(XPRS_LPSTOP, z)
savebasis(bas)
minimize(XPRS_CONT, z)
```

6.2.3 Submodel 2: fixed integer variables

The second step of our decomposition algorithm consists in solving a subproblem where all integer variables are fixed to their solution values found in the first step. The structure of the model implementing this step is quite similar to the previous submodel. When the model is run, it reads the invariant data from memory and sets up the objective function. It then halts at the line `wait` at the beginning of the loop to wait for a step 2 solving event sent by the parent model. At every solving iteration the constraints `CTR` are redefined using the coefficients read from memory and the solution is written back to memory. Below follows the source of this model (file `benders_cont.mos`).

```
model "Benders (continuous problem)"
 uses "mmxprs", "mmjobs"

 parameters
  NCTVAR = 3
  NINTVAR = 3
  NC = 4
  BIGM = 1000
 end-parameters

 declarations
  STEP_0=2                          ! Event codes sent to submodels
  STEP_2=4
  STEP_3=5
  EVENT_SOLVED=6                    ! Event codes sent by submodels
  EVENT_READY=8

  CtVars = 1..NCTVAR                ! Continuous variables
  IntVars = 1..NINTVAR              ! Discrete variables
  Ctrs = 1..NC                      ! Set of constraints (orig. problem)
```

```
  A: array(Ctrs,CtVars) of integer    ! Coeff.s of continuous variables
  B: array(Ctrs,IntVars) of integer   ! Coeff.s of discrete variables
  b: array(Ctrs) of integer           ! RHS values
  C: array(CtVars) of integer         ! Obj. coeff.s of continuous variables
  Ctr: array(Ctrs) of linctr          ! Constraints of orig. problem

  sol_u: array(Ctrs) of real          ! Solution of dual problem
  sol_x: array(CtVars) of real        ! Solution of primal prob. (cont.)
  sol_y: array(IntVars) of real       ! Solution of primal prob. (integers)

  x: array(CtVars) of mpvar           ! Continuous variables
 end-declarations

 initializations from "bin:shmem:probdata"
  A  B  b  C
 end-initializations

 Obj:= sum(i in CtVars) C(i)*x(i)

 send(EVENT_READY,0)                  ! Model is ready (= running)

! (Re)solve this model until it is stopped by event "STEP_3"
 repeat
  wait
  dropnextevent

  initializations from "bin:shmem:sol"
   sol_y
  end-initializations

  forall(j in Ctrs)
   Ctr(j):= sum(i in CtVars) A(j,i)*x(i) +
            sum(i in IntVars) B(j,i)*sol_y(i) >= b(j)

  minimize(Obj)                       ! Solve the problem

 ! Store values of u and x
  forall(j in Ctrs) sol_u(j):= getdual(Ctr(j))
  forall(i in CtVars) sol_x(i):= getsol(x(i))

  initializations to "bin:shmem:sol"
   sol_u  sol_x
  end-initializations

  send(EVENT_SOLVED, getobjval)

  write_("Step 2: ", getobjval, "\n  u: ")
  forall(j in Ctrs) write(sol_u(j), " ")
  write("\n  x: ")
  forall(i in CtVars) write(getsol(x(i)), " ")
  writeln
  fflush

 until false

end-model
```

6.2.4 Submodel 0: start solution

To start the decomposition algorithm we need to generate an initial set of values for the continuous variables. This can be done by solving the dual problem in the continuous variables with a dummy objective function. A second use of the dual problem is for Step 2 of the algorithm, replacing the primal model we have seen in the previous section. The implementation of this submodel takes into account these two cases: within the solving loop we test for the type (class) of event that has been sent by the

parent problem and choose the problem to be solved accordingly.

The main part of this model is implemented by the following Mosel code (file `benders_dual.mos`).

```
model "Benders (dual problem)"
 uses "mmxprs", "mmjobs"

 parameters
  NCTVAR = 3
  NINTVAR = 3
  NC = 4
  BIGM = 1000
 end-parameters

 forward procedure save_solution

 declarations
  STEP_0=2                            ! Event codes sent to submodels
  STEP_2=4
  EVENT_SOLVED=6                      ! Event codes sent by submodels
  EVENT_INFEAS=7
  EVENT_READY=8

  CtVars = 1..NCTVAR                  ! Continuous variables
  IntVars = 1..NINTVAR                ! Discrete variables
  Ctrs = 1..NC                        ! Set of constraints (orig. problem)

  A: array(Ctrs,CtVars) of integer    ! Coeff.s of continuous variables
  B: array(Ctrs,IntVars) of integer   ! Coeff.s of discrete variables
  b: array(Ctrs) of integer           ! RHS values
  C: array(CtVars) of integer         ! Obj. coeff.s of continuous variables

  sol_u: array(Ctrs) of real          ! Solution of dual problem
  sol_x: array(CtVars) of real        ! Solution of primal prob. (cont.)
  sol_y: array(IntVars) of real       ! Solution of primal prob. (integers)

  u: array(Ctrs) of mpvar             ! Dual variables
 end-declarations

 initializations from "bin:shmem:probdata"
  A  B  b  C
 end-initializations

 forall(i in CtVars) CtrD(i):= sum(j in Ctrs) u(j)*A(j,i) <= C(i)

 send(EVENT_READY,0)                  ! Model is ready (= running)

! (Re)solve this model until it is stopped by event "STEP_3"
 repeat
  wait
  ev:= getnextevent
  Alg:= getclass(ev)

  if Alg=STEP_0 then                  ! Produce an initial solution for the
                                      ! dual problem using a dummy objective
   maximize(sum(j in Ctrs) u(j))
   if(getprobstat = XPRS_INF) then
    writeln_("Problem is infeasible")
    send(EVENT_INFEAS,0)              ! Problem is infeasible
   else
    write_("**** Start solution: ")
    save_solution
   end-if

  else                                ! STEP 2: Solve the dual problem for
                                      ! given solution values of y
   initializations from "bin:shmem:sol"
```

```
    sol_y
   end-initializations

   Obj:= sum(j in Ctrs) u(j)* (b(j) - sum(i in IntVars) B(j,i)*sol_y(i))
   maximize(XPRS_PRI, Obj)

   if(getprobstat=XPRS_UNB) then
    BigM:= sum(j in Ctrs) u(j) <= BIGM
    maximize(XPRS_PRI, Obj)
   end-if

   write_("Step 2: ")
   save_solution                      ! Write solution to memory
   BigM:= 0                           ! Reset the 'BigM' constraint
  end-if

 until false
```

This model is completed by the definition of the subroutine `save_solution` that writes the solution to memory and informs the parent model of it being available by sending the `EVENT_SOLVED` message.

```
! Process solution data
 procedure save_solution
 ! Store values of u and x
  forall(j in Ctrs) sol_u(j):= getsol(u(j))
  forall(i in CtVars) sol_x(i):= getdual(CtrD(i))

  initializations to "bin:shmem:sol"
   sol_u  sol_x
  end-initializations

  send(EVENT_SOLVED, getobjval)

  write(getobjval, "\n  u: ")
  forall(j in Ctrs) write(sol_u(j), " ")
  write("\n  x: ")
  forall(i in CtVars) write(getdual(CtrD(i)), " ")
  writeln
  fflush
 end-procedure

end-model
```

6.3 Alternative implementation with multiple problems

Since our decomposition algorithm is formed by a series of sequential optimization runs it is possible to implement the whole approach as a single model defining several *problems* (instead of several *model files*, each with a single problem). This alternative implementation requires significantly less effort for data handling (problems simply share data) and coordination of solving (problem solving in a single model always is sequential). As a result, the total Mosel model code is shorter. However, by collecting all problems into a single model, the testing of (sub)problems in isolation is made somewhat more complicated—in the previous version we can simply run one of the (sub)models as a stand-alone model, a feature that certainly is helpful when developing large applications.

6.3.1 Main problem

In the (main) model, all code related to (sub)model handling is replaced by the somewhat simpler handling of (sub)problems that are defined and solved within the same model as the main problem. Here we just display the part of the declarations that are new or modified, followed by the setup of the subproblems and the (unchanged) solution algorithm with the definition of the 'start solve' subroutines.

For every submodel we now have two new subroutines, `define_*prob` and `solve_*prob` that replace the separate model files. These subroutines are listed in the following sections.

```
 declarations
  Obj: linctr                          ! Continuous objective
  y: array(IntVars) of mpvar           ! Discrete variables
  z: mpvar                             ! Objective function variable
  u: array(Ctrs) of mpvar              ! Dual variables
  x: array(CtVars) of mpvar            ! Continuous variables

  RM: range                            ! Problem indices
  stepprob: array(RM) of mpproblem     ! Subproblems
  status: array(mpproblem) of integer ! Subproblem status
 end-declarations

! **** Subproblems ****
                                       ! Create and define submodels
 create(stepprob(1)); define_intprob(stepprob(1))

 if ALG=1 then
  create(stepprob(2)); define_dualprob(stepprob(2))
 else
  create(stepprob(0)); define_dualprob(stepprob(0))
  create(stepprob(2)); define_contprob(stepprob(2))
 end-if

! **** Solution algorithm ****

 start_solution                        ! 0. Initial solution for cont. var.s
 ct:= 1
 repeat
 writeln_("\n**** Iteration: ", ct)
  solve_primal_int(ct)                 ! 1. Solve problem with fixed cont.
  solve_cont                           ! 2. Solve problem with fixed int.
  ct+=1
 until eval_solution                   !    Test for optimality
 print_solution                        ! 3. Retrieve and display the solution

!-------------------------------------------------------------
! Produce an initial solution for the dual problem using a dummy objective
 procedure start_solution
  num:= if(ALG=1, 2, 0)
  res:=solve_dualprob(stepprob(num), STEP_0)  ! Start the problem solving

  if status(stepprob(num))=STAT_INFEAS then
   writeln_("Problem is infeasible")
   exit(6)
  end-if
 end-procedure

!-------------------------------------------------------------
! Solve a modified version of the primal problem, replacing continuous
! variables by the solution of the dual
 procedure solve_primal_int(ct: integer)
  sol_obj:= solve_intprob(stepprob(1), ct)
 end-procedure

!-------------------------------------------------------------
! Solve the Step 2 problem (dual or primal depending on value of ALG)
! for given solution values of y
 procedure solve_cont
  if ALG=1 then                        ! Start the problem solving
   res:= solve_dualprob(stepprob(2), STEP_2)
  else
   res:= solve_contprob(stepprob(2))
  end-if
 end-procedure
```

6.3.2 Subproblem 1: fixed continuous variables

The bounds and integrality conditions for the integer subproblem are stated once (`define_intprob`). Each time the integer subproblem is solved (`solve_intprob`) a new constraint gets added. The solving routine also stores the solution values and the problem status into global data structures.

```
! Define the integer problem
 procedure define_intprob(prob:mpproblem)
  with prob do
   z is_free                          ! Objective is a free variable
   forall(i in IntVars) y(i) is_integer ! Integrality condition
   forall(i in IntVars) y(i) <= BIGM    ! Set (large) upper bound
  end-do
  status(prob):= STAT_READY
 end-procedure

!-------------------------------------------------------------
! Solve the integer problem
 function solve_intprob(prob:mpproblem, ct:integer): real
  with prob do
   status(prob):= STAT_READY

   ! Add a new constraint
   MC(ct):= z >= sum(i in IntVars) D(i)*y(i) +
                 sum(j in Ctrs) sol_u(j)*(b(j) - sum(i in IntVars) B(j,i)*y(i))

   minimize(z)

   ! Store solution values of y
   forall(i in IntVars) sol_y(i):= getsol(y(i))

   returned:=getobjval
   status(prob):= STAT_SOLVED

   write_("Step 1: ", getobjval, "\n  y: ")
   forall(i in IntVars) write(sol_y(i), " ")
   write_("\n  Slack: ")
   forall(j in 1..ct) write(getslack(MC(j)), " ")
   writeln
   fflush

  end-do
 end-function
```

6.3.3 Subproblem 2: fixed integer variables

The setup routine for the continuous problem (`define_contprob`) only defines the objective function. The constraints are re-defined each time the problem gets solved (`solve_contprob`).

```
! Define the continuous primal problem
 procedure define_contprob(prob:mpproblem)
  with prob do
   Obj:= sum(i in CtVars) C(i)*x(i)
  end-do
  status(prob):= STAT_READY
 end-procedure

!-------------------------------------------------------------
! Solve the continuous problem
 function solve_contprob(prob:mpproblem): real
  with prob do
   status(prob):= STAT_READY

  ! (Re)define and solve this problem
```

```
   forall(j in Ctrs)
    Ctr(j):= sum(i in CtVars) A(j,i)*x(i) +
             sum(i in IntVars) B(j,i)*sol_y(i) >= b(j)

   minimize(Obj)                        ! Solve the problem

  ! Store values of u and x
   forall(j in Ctrs) sol_u(j):= getdual(Ctr(j))
   forall(i in CtVars) sol_x(i):= getsol(x(i))

   returned:=getobjval
   status(prob):= STAT_SOLVED

   write_("Step 2: ", getobjval, "\n  u: ")
   forall(j in Ctrs) write(sol_u(j), " ")
   write("\n  x: ")
   forall(i in CtVars) write(getsol(x(i)), " ")
   writeln
   fflush

  end-do
 end-function
```

6.3.4 Subproblem 0: start solution

The setup routine for the dual problem (`define_dualprob`) states the constraints for this model. The solving subroutines `solve_dualprob` implements two cases, depending on whether we use this model with a dummy objective for finding a start solution (*Step 0*) or in the second part of the solution algorithm where the objective is defined with the current solution values (*Step 2*).

```
! Define the dual problem
 procedure define_dualprob(prob:mpproblem)
  with prob do
   forall(i in CtVars) CtrD(i):= sum(j in Ctrs) u(j)*A(j,i) <= C(i)
  end-do
  status(prob):= STAT_READY
 end-procedure

!-------------------------------------------------------------
! (Re)solve the dual problem
 function solve_dualprob(prob:mpproblem, Alg:integer): real
  with prob do
   status(prob):= STAT_READY

   if Alg=STEP_0 then                 ! Produce an initial solution for the
                                      ! dual problem using a dummy objective
    maximize(sum(j in Ctrs) u(j))
    if(getprobstat = XPRS_INF) then
     writeln_("Problem is infeasible")
     status(prob):= STAT_INFEAS       ! Problem is infeasible
    else
     write_("**** Start solution: ")
     save_dualsolution(prob)
     returned:= getobjval
    end-if

   else                               ! STEP 2: Solve the dual problem for
                                      ! given solution values of y
    Obj:= sum(j in Ctrs) u(j)* (b(j) - sum(i in IntVars) B(j,i)*sol_y(i))
    maximize(XPRS_PRI, Obj)

    if(getprobstat=XPRS_UNB) then
     BigM:= sum(j in Ctrs) u(j) <= BIGM
     maximize(XPRS_PRI, Obj)
    end-if
```

```
    write_("Step 2: ")
    save_dualsolution(prob)           ! Save solution values
    returned:= getobjval
    BigM:= 0                          ! Reset the 'BigM' constraint
   end-if

  end-do
 end-function
```

Similarly to the previous implementation as a separate model, we have moved the storing of the solution values into a subroutine (`save_dualsolution`). Notice that it is not necessary to switch problems using `with prob do` in this subroutine because it gets called from within a subproblem.

```
! Process dual solution data
 procedure save_dualsolution(prob:mpproblem)
 ! Store values of u and x
  forall(j in Ctrs) sol_u(j):= getsol(u(j))
  forall(i in CtVars) sol_x(i):= getdual(CtrD(i))

  status(prob):= STAT_SOLVED

  write(getobjval, "\n  u: ")
  forall(j in Ctrs) write(sol_u(j), " ")
  write("\n  x: ")
  forall(i in CtVars) write(getdual(CtrD(i)), " ")
  writeln
  fflush
 end-procedure
```

6.4 Results

The optimal solution to our small test problem has the objective function value 18.1852. Our program produces the following output, showing that the problem is solved to optimality with 3 iterations (looping around steps 1 and 2) of the decomposition algorithm:

```
**** Start solution: 4.05556
  u: 0.740741 1.18519 2.12963 0
  x: 0.611111 0.166667 0.111111

**** Iteration: 1
Step 1: -1146.15
  y: 1000 0 0
  Slack: 0
Step 2: 1007
  u: 0 1 0 0
  x: 0 251.75 0
Test optimality: -3146.15 = 1007 : false

**** Iteration: 2
Step 1: 17.0185
  y: 3 0 0
  Slack: 0 -1.01852
Step 2: 12.5
  u: 0 1 2.5 0
  x: 0.5 2.5 0
Test optimality: 11.0185 = 12.5 : false

**** Iteration: 3
Step 1: 18.1852
  y: 2 0 0
  Slack: 0 -5.18519 -0.185185
Step 2: 14.1852
  u: 0.740741 1.18519 2.12963 0
```

```
  x: 1.03704 2.22222 0.037037
Test optimality: 14.1852 = 14.1852 : true

**** Solution (Benders): 18.1852
  x: 1.03704 2.22222 0.037037   y: 2 0 0
```

7 Start solution heuristic: Concurrent model clones using shared data structures

The example in this section shows how to use the concept of cloned models sharing data structures introduced in Section 2.7.3 for the implementation of a start solution heuristic for the classical jobshop scheduling problem. The generation of the start solutions involves concurrent solving of several instances of a subproblem that uses the same input data as the main problem and needs to communicate back the detailed results.

7.1 Example problem: jobshop scheduling

We wish to schedule the production of a set of jobs on a set of machines. Every job is produced by a sequence of tasks, each of these tasks is processed on a different machine with a specific given duration. A machine processes at most one job at a time. Tasks are non-preemptive (once started the processing cannot be interrupted). The objective is to minimize the total duration of the schedule, that is, the completion time of the last task, also referred to as *makespan*.

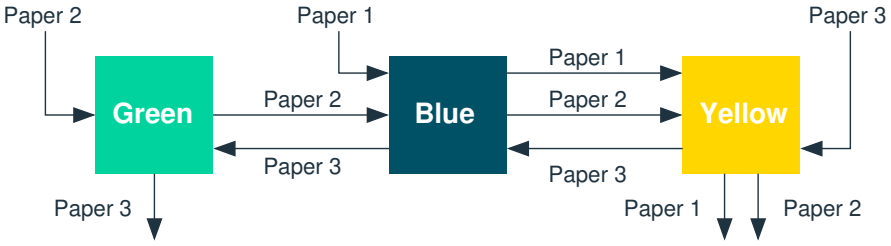


Figure 8: Jobshop example with 3 jobs (wallpapers) and 3 machines (colors)

This combinatorial problem is notoriously difficult to solve to optimality with mathematical programming approaches even for relatively small data instances. For practical purposes it is therefore quite a common practice to work with heuristics or some form of incomplete search, for example by stopping the solution algorithms after a certain laps of time. In such a case it is particularly important to be able to quickly compute good solutions and lower bound estimates.

Our start solution heuristic for the jobshop problem relies on the following idea: By solving single machine subproblems (sequencing the tasks that are processed on the same machine) we can generate partial solutions to the full jobshop problem that are loaded as start solutions into the MIP optimizer. This approach requires us to state and solve two distinct problems:

1. Several instances (one per machine) of a sequencing problem—these subproblems are independent and can therefore be formulated and solved concurrently
2. One instance of the full jobshop problem

7.1.1 Jobshop scheduling model

For the formulation of the jobshop problem we work with a set of *JOBS*, each composed of a series of *TASKS* that represent the processing of a job j on a specific resource m (from the set *RESOURECS*) with a known duration DUR_{jm}. The decision variables are the start and completion times of the tasks (start_{jt} and comp_{jt} respectively) and a set of binary variables y_{mij} for stating the sequence of tasks from two different jobs i and j on a resource m.

The objective function variable $makespan$ is constrained to take the value of the latest completion time. We further need to state the order of tasks within every job (precedence relations) and the sequence of tasks on a given resouces (disjunctions between tasks).

Objective:

$$\text{minimize } makespan$$

Decision variables:

$$\forall j \in JOBS, t \in TASKS : start_{jt}, comp_{jt} \in [0, MAXTIME]$$
$$\forall m \in RESOURCES, i < j \in JOBS : y_{mij} \in \{0, 1\}$$

Precedence relations:

$$\forall j \in JOBS : makespan \geqslant start_{j,NBRES} + DUR_{j,NBRES}$$
$$\forall j \in JOBS, t \in TASKS - \{NBRES\} : start_{jt} + DUR_{jt} \leqslant start_{j,t+1}$$

Disjunctions between tasks on same machine:

$$\forall m \in RESOURCES, i < j \in JOBS : \quad start_{i,RES_{mi}} + DUR_{i,RES_{mi}} \leqslant start_{j,RES_{mj}} + M \cdot y_{mij}$$
$$start_{j,RES_{mj}} + DUR_{j,RES_{mj}} \leqslant start_{i,RES_{mi}} + M \cdot (1 - y_{mij})$$

Linking start and completion times:

$$\forall j \in JOBS, t \in TASKS : start_{jt} + DUR_{jt} = comp_{jt}$$

7.1.2 Single machine sequencing subproblem

The sequencing problem on a specific machine m is expressed via a separate set of binary decision variables $rank_{jk}$ that indicate whether the task from job j is processed in position k on this resource. The start time of the task in position k is expressed by $tstart_k$. We take into account the release date REL_j (sum of the durations of preceding tasks) and the total duration (processing time DUR_{jm} of the task itself and duration $DURSUCC_j$ of all its successors within the job) for calculating a job completion time $jcomp_j$, the maximum of which (= $makespan$) is to be minimized. This $makespan$ of the individual subproblems provides a lower bound on the makespan of the full jobshop problem.

Objective:

$$\text{minimize } makespan$$

Decision variables:

$$\forall j, k \in JOBS : rank_{jk} \in \{0, 1\}$$
$$\forall j \in JOBS : jcomp_j \in [0, MAXTIME]$$
$$\forall k \in JOBS : tstart_k \in [0, MAXTIME]$$

One job per position, one position per job:

$$\forall k \in JOBS : \sum_{j \in JOBS} rank_{jk} = 1$$
$$\forall j \in JOBS : \sum_{k \in JOBS} rank_{jk} = 1$$

Sequence of jobs:

$$\forall k \in 1, ..., NJOBS - 1 : tstart_{k+1} \geqslant tstart_k + \sum_{j \in JOBS} DUR_{jm} \cdot rank_{jk}$$

Release dates for start times:

$$\forall k \in JOBS : tstart_k \geqslant \sum_{j \in JOBS} REL_j \cdot rank_{jk}$$

Completion times:

$$\forall k \in JOBS : \quad jcomp_k = tstart_k + \sum_{j \in JOBS} (DUR_{jm} + DURSUCC_j) \cdot rank_{jk}$$
$$makespan \geqslant jcomp_k$$

7.2 Implementation

The implementation of a model that starts submodels in the form of clones in order to work with shared data structures needs to contain the code for both, controlling model and submodels, in a single file.

The main model logic with all declarations of entities and subroutines is shown below: the actual model is contained in the last `if` statement that decides which subroutines to execute depending on whether the model is run as the controlling model (solving the full scheduling problem) or as a clone that solves a sequencing subproblem.

```
model "Job shop - Clone"
 uses "mmxprs", "mmjobs", "mmsystem"

 parameters
  DATAFILE = "mt10.dat"           ! Muth and Thompson 10x10 instance
  MACH = 1                        ! Subproblem identifier
  MAXTHRD = 2                     ! Max number of parallel threads for one inst.
 end-parameters

 declarations
 ! **** Shared data
  NBJOBS: shared integer                      ! Number of jobs
  NBRES: shared integer                       ! Number of resources
  JOBS: shared range                          ! Set of jobs
  TASKS: shared range                         ! Set of tasks (one per machine)
  TASKSM1: shared range                       ! Set of tasks without last
  RESOURCES: shared range                     ! Set of resources
  DUR: shared array(JOBS,TASKS) of integer          ! Durations of tasks
  RESIND: shared array(RESOURCES,JOBS) of integer   ! Task index per resource
  MAXDUR: shared array(RESOURCES) of integer
  BIGM: integer                               ! Used in formulation of main prob

  cutoff,starttime: real                      ! Temp. value used in main problem
  bbindex: integer                            ! Temp. value used in main problem

 ! **** Declarations for main problem
  start: array(JOBS,TASKS) of mpvar           ! Start times of tasks
  comp: array(JOBS,TASKS) of mpvar            ! Completion times of tasks
  makespan: mpvar                             ! Schedule completion time
  y: dynamic array(RESOURCES,JOBS,JOBS) of mpvar  ! Disjunction variables

 ! **** Declarations for sequencing subproblems
  heursol: dynamic array(set of mpvar) of real
  rank: array(JOBS,JOBS) of mpvar             ! =1 if job j at position k
  jcomp,tstart: array(JOBS) of mpvar          ! Start time of job at position k
```

```
 ! **** Model and solution handling
  SeqMod: array(RESOURCES) of Model             ! Models
  NOSOL=2                                       ! "No sol found" event
  NEWSOL=3                                      ! "Solution found" event
  pos: shared array(RESOURCES,JOBS) of integer  ! Subproblem solutions
  Makespan: shared array(RESOURCES) of real     ! Subproblem makespan

 ! **** Subroutines
  procedure readdata
  procedure sequencemachine(m:integer)
  procedure disjunctiveconstraints(m:integer)
  procedure solvemain
  procedure printsol
 end-declarations

!*******************************Model body*******************************

 if getparam("sharingstatus")<=0 then
   readdata
   solvemain
   printsol
 else
   sequencemachine(MACH)
 end-if

end-model
```

7.2.1 Jobshop problem

The controlling model starts with the data input via the following subroutine `readdata`, all input data items are a marked as `shared` meaning that they are available within clones of this model:

```
 procedure readdata
   initializations from DATAFILE
     NBJOBS NBRES
   end-initializations
   JOBS := 1..NBJOBS; finalize(JOBS)
   TASKS := 1..NBRES ; finalize(TASKS)
   TASKSM1 := 1..NBRES-1  ; finalize(TASKSM1)
   RESOURCES := 0..NBRES-1 ; finalize(RESOURCES)

   declarations
     taskUse: array(JOBS,TASKS) of integer   ! Resource use of tasks
   end-declarations

   initializations from DATAFILE
     taskUse  DUR as "taskDuration"
   end-initializations

   forall(m in RESOURCES,j in JOBS,t in TASKS | taskUse(j,t) = m) RESIND(m,j):=t

   BIGM:=sum(j in JOBS,t in TASKS) DUR(j,t)    ! Some (sufficiently) large value
   forall(m in RESOURCES) MAXDUR(m):=sum(j in JOBS) DUR(j,RESIND(m,j))
 end-procedure
```

The implementation of the jobshop problem is contained in the following subroutine that defines the problem, creates and runs the concurrent sequencing subproblems as clones, and transforms the solutions returned by these into the required from for adding them as (partial) start solutions into the full problem.

```
 procedure solvemain
  ! Precedence constraints between last task of every job and makespan
   forall(j in JOBS) makespan >= start(j,NBRES) + DUR(j,NBRES)
```

```
  ! Precedence constraints between the tasks of every job
   forall(j in JOBS,t in TASKSM1)
     start(j,t) + DUR(j,t) <= start(j,t+1)

  ! Disjunctions
   forall(r in RESOURCES) disjunctiveconstraints(r)

  ! Linking start and completion times
   forall(j in JOBS,t in TASKS) start(j,t)+DUR(j,t) = comp(j,t)

  ! Bound on latest completion time
   makespan <= BIGM

   starttime:=gettime

  ! Solve the single machine sequencing problems
   forall(m in RESOURCES) do
     load(SeqMod(m))                      ! Clone the current model
     SeqMod(m).uid:= m
     setworkdir(SeqMod(m), ".")
     run(SeqMod(m), "MACH="+m +",MAXTHRD="+1)
   end-do
   tct:=0
   while (tct<NBRES) do
     wait                                 ! Wait for the next event
     Msg:= getnextevent                   ! Get the event
     mid:= Msg.fromuid                    ! Get model number of sender
     if getclass(Msg)=NEWSOL then         ! Get the event class
       if Makespan(mid)>cutoff then
         cutoff:= Makespan(mid);
         bbindex:=mid
       end-if
       writeln_("*** (", gettime-starttime, "s) Machine ", mid, ": ", Makespan(mid))

       forall(i,j in JOBS | i<j ) heursol(y(mid,i,j)):= if(pos(mid,i)<pos(mid,j), 0, 1)
       loadprob(makespan)                 ! Make sure problem is loaded
       id:="mach"+mid
       addmipsol(id,heursol)              ! Load the heuristic solution
       writeln_("Loaded solution Id: ", id)
       setcallback(XPRS_CB_SOLNOTIFY, ->solnotify)
       delcell(heursol)                   ! Delete saved solution values
     elif getclass(Msg)=NOSOL then
       writeln_("Subproblem ", mid, " has no solution. Stopping")
       exit(1)
     else
       tct+=1
       unload(SeqMod(mid))                ! Delete subproblem
     end-if
   end-do

   writeln_("(", gettime-starttime, "s) Best lower bound: ", cutoff)

  ! Re-inforce use of user solutions by local search MIP heuristics
   setparam("XPRS_USERSOLHEURISTIC", 3)

  ! Set a time limit
   setparam("XPRS_SOLTIMELIMIT", 5)

  ! Solve the problem: minimize latest completion time
   minimize(makespan)
 end-procedure

! **** Callback reporting the status of loaded solutions ****
 procedure solnotify(id:string, status:integer)
   writeln_("Optimiser loaded solution ", id, " status=", status)
 end-procedure
```

The subroutine `disjunctiveconstraints` defines the disjunctions between tasks on the same resource—by moving the definition of these constraints into a separate subroutine it becomes easy to experiment with alternative formulations for these constraints (such as by using indicator constraints in place of the big-M construct employed in the version shown here).

```
procedure disjunctiveconstraints(m:integer)
  forall(i,j in JOBS | i<j) do
    create(y(m,i,j))
    y(m,i,j) is_binary               !=1 if i>>j, =0 if i<<j

    start(i,RESIND(m,i))+DUR(i,RESIND(m,i)) <=
      start(j,RESIND(m,j))+BIGM*y(m,i,j)
    start(j,RESIND(m,j))+DUR(j,RESIND(m,j)) <=
      start(i,RESIND(m,i))+BIGM*(1-y(m,i,j))
   end-do
end-procedure
```

The implementation of the jobshop problem is completed with some solution display at the end of the solving of the full problem:

```
procedure printsol
  writeln_("(", gettime-starttime, "s) Total completion time: ", getobjval)
  forall(j in JOBS) write(strfmt(j,8))
  writeln
  forall(m in RESOURCES) do
    write(strfmt((m+1),-3))
    forall(j in JOBS)
      if(DUR(j,RESIND(m,j))>0) then
        write(formattext("%4d-%3d", round(getsol(start(j,RESIND(m,j)))),
             round(getsol(start(j,RESIND(m,j)))+DUR(j,RESIND(m,j)))))
      else
        write(strfmt(" ",6))
      end-if
    writeln
  end-do
end-procedure
```

7.2.2 Sequencing subproblem

The implementation of the sequencing subproblem is provided by the following subroutine that terminates by sending an event that notifies the controlling model of the solving status (indicating whether a subproblem solution is available):

```
procedure sequencemachine(m:integer)
 ! One job per position
  forall(k in JOBS) sum(j in JOBS) rank(j,k) = 1

 ! One position per job
  forall(j in JOBS) sum(k in JOBS) rank(j,k) = 1

 ! Sequence of jobs
  forall(k in 1..NBJOBS-1)
    tstart(k+1) >=
      tstart(k) + sum(j in JOBS) DUR(j,RESIND(m,j))*rank(j,k)

 ! Start times (release date = min total duration for preceding tasks)
  forall(j in JOBS) REL(j):= sum(t in 1..RESIND(m,j)-1) DUR(j,t)
  forall(j in JOBS) DURSUCC(j):= sum(t in RESIND(m,j)+1..NBRES) DUR(j,t)

  forall(k in JOBS) tstart(k) >= sum(j in JOBS) REL(j)*rank(j,k)

 ! Completion times
  forall(k in JOBS) jcomp(k) =
```

```
    tstart(k) + sum(j in JOBS) (DUR(j,RESIND(m,j))+DURSUCC(j))*rank(j,k)

  forall(j,k in JOBS) rank(j,k) is_binary

 ! Objective function: minimize latest completion time
  forall(k in JOBS) makespan >= jcomp(k)

 ! Solve the problem
  minimize(makespan)

 ! Solution reporting
  if getprobstat=XPRS_OPT then
   ! writeln_("*** Machine ", m, ": ", getobjval)
    forall(j in JOBS) pos(m,j):= round(sum(k in JOBS) k*rank(j,k).sol)
    Makespan(m):=getobjval

   ! Send "solution found" signal
    send(NEWSOL, getobjval)
  else
    send(NOSOL, 0)
  end-if
end-procedure
```

7.3 Results

For the datasets used for testing (such as the much studied MT10 instance from [**?**]) most often the MIP heuristics are only able to complete few of the partial solutions to a full solution, but where this is possible this solution usually is better than any initial solution detected by the default algorithms.

Several other implementation variants are available for this start solution algorithm:

1. Instead of starting model clones for solving the sequencing subproblems these could simply be treated as **multiple problems** that are **solved sequentially** within a single model. All problems and data simply reside within a single model in this case, so there is no need to employ any mechanism for sharing data. Furthermore, only a single subproblem is active in the solver at any time, hence limiting the amount of resouces (memory, licenses) that are in concurrent use. An implementation example is provided in the file `jobshopas.mos`.
2. As for the examples discussed in the preceding sections of this paper, **concurrent solving** with submodels can be implemented with a **separate file** for every problem (so one main model file, `jobshopasp.mos`, and one submodel file, `jobseq.mos`, that is loaded for each subproblem instance to be processed in parallel). In this case, data are exchanged between the parent model and its descendants via the shared memory I/O driver of *mmjobs*. As opposed to the implementation relying on model clones this version could be extended to work with several distinct Mosel instances for processing the submodels.

With all three implementation variants we are solving the same sequencing subproblems and will therefore obtain the same (partial) start solutions. When solving subproblems concurrently our implementation processes the results as soon as they become available, so the order with which start solutions are added and as a consequence the order in which they are processed by the solver is to some extend non-deterministic and may result in diverging behaviour of the solution algorithms. With sequential execution the order of adding the user solution information is always the same.

8 Summary

The examples in this white paper show how to use the functionality provided by the *mmjobs* module. Without giving an exhaustive overview of the technical possibilities they provide starting points for

implementation of, and experimentation with, parallel solving and other multi-problem solution approaches.

Any solver module available for Mosel may be used in conjunction with *mmjobs*. However, parallel solving of multiple problems with the same solver is only possible if the underlying solver can work in a multi-threaded environment. This is the case for Xpress Optimizer and Xpress NonLinear as well as for Xpress Kalis.