AIMMS Language Reference - Solving Mathematical Programs

Aimms 4

This documentation was typeset by AIMMS B.V. using LATEX and the LUCIDA font family.

Chapter 15

Solving Mathematical Programs

Mathematical program components

A *mathematical program* consists of

- a set of unknowns to be determined,
- a collection of constraints that has to be satisfied, and
- an (optional) objective function to be optimized.

The aim of a mathematical program is to find a solution with the aid of a solver such that the objective function assumes an optimal (i.e. minimal or maximal) value.

Different types

Depending on the characteristics of the variables and constraints, a mathematical program in AIMMS can be classified as one of the following.

- If the objective function and all constraints contain only linear expressions (in terms of the variables), and all variables can assume continuous values within their ranges, then the program is a *linear* program.
- If some of the variables in a linear program can assume only integer values, then the program is a *linear mixed integer* program.
- If the objective is a quadratic function in terms of the variables while the constraints are linear, then the program is a *quadratic* program.
- If the objective is neither linear nor quadratic, or some of the constraints contain nonlinear expressions, the program is a *nonlinear* program.

AIMMS will automatically call the appropriate solver to find an (optimal) solution.

This chapter

This chapter first discusses the declaration of a mathematical program, together with auxiliary functions that you can use to specify its set of variables and constraints. The `SOLVE` execution statement needed to solve any type of mathematical program is presented, and, finally, AIMMS' capabilities to help resolve infeasibilities in your model are discussed.

15.1 `MathematicalProgram` declaration and attributes

Attributes

The attributes of mathematical programs are listed in Table 15.1.

Attribute	**Value-type**	**See also page**
`Objective`	*variable-identifier*	
`Direction`	`minimize`, `maximize`	
`Variables`	*variable-set*	
`Constraints`	*constraint-set*	
`Type`	*model-type*	
`ViolationPenalty`	*reference*	243
`Text`	*string*	19
`Comment`	*comment string*	19
`Convention`	*convention*	536

Table 15.1: `MathematicalProgram` attributes

Example

The following example illustrates a typical mathematical program.

```
MathematicalProgram TransportModel {
    Objective   : TransportCost;
    Direction   : minimize;
    Constraints : AllConstraints;
    Variables   : AllVariables;
    Type        : lp;
}
```

It defines the linear program `TransportModel`, which is built up from all constraints and variables in the model text. The variable `TransportCost` serves as the objective function to be minimized.

The `Objective` attribute

With the `Objective` attribute you can specify the objective of your mathematical program. Its value must be a reference to a (defined) variable or any other variable expression. When you want to use the objective value in the end-user interface of your model, the `Objective` attribute must be a variable reference.

Omitting the objective

If you do not specify an objective, your mathematical program will be solved to find a feasible solution and it will then terminate.

The `Direction` attribute

In conjunction with an objective you must use the `Direction` attribute to indicate whether the solver should `minimize` or `maximize` the objective. During a `SOLVE` statement you can override this direction by using a `WHERE` clause for the `direction` option.

The `Variables` attribute

With the `Variables` attribute you can specify which set of variables are to be included in your mathematical program. Its must be either the predefined set `AllVariables` or a subset thereof. The set `AllVariables` is predefined by AIMMS, and it contains the names of all the variables declared in your model. Its contents cannot be changed. If you mathematical program contains an objective, AIMMS will automatically add this to set of generated variables during generation.

Variables as parameters

If the `Variables` attribute is assigned a subset of the set `AllVariables`, AIMMS will treat all the variables outside this set as if they were parameters. That is, all occurrences of such variables will not result in the generation of individual variables for the solver, but will be accounted for in the right-hand side of the constraint according to their value during generation.

Compare to `NonvarStatus`

The `Variables` attribute performs a similar function as the `NonvarStatus` attribute or the `.NonVar` suffix of a variable (see also Section 14.1). The `Variables` attribute in a mathematical program allows you to quickly change the status of an entire class of variables, while the `NonvarStatus` (in a variable declaration) gives much finer control at the individual level. As shown below, the latter is very useful to perform model algebra.

The `Constraints` attribute

With the `Constraints` attribute you can specify which constraints are part of your mathematical program. Its value must be either the predefined set `AllConstraints` or a subset thereof. The set `AllConstraints` contains the names of all declared constraints plus the names of all variables which have a definition attribute. Its contents is computed at compile time, and cannot be changed.

- If you specify the set `AllConstraints`, AIMMS will generate individual constraints for all declared constraints and variables with a definition.
- If you specify a subset of the set `AllConstraints`, AIMMS will only generate individual constraints for the declared constraints and defined variables in that subset.

If you mathematical program has an objective which is a defined variable, its definition is automatically added to the set of generated constraints during generation.

Defined variables

Variables with a nonempty definition attribute have a somewhat special status. Namely, for every defined variable AIMMS will not only generate this variable, but will also generate a constraint containing its definition. Therefore, defined variables are contained in both the predefined sets `AllVariables` and `AllConstraints`. You can add a defined variable to the variable and constraint set of a mathematical program independently.

- If you omit a defined variable from the variable set of a mathematical program, all occurrences of the variable will be fixed to its current value and accounted for in the right-hand side of all constraints.
- If you omit a defined variable from the constraint set of a mathematical program, the defining constraint will not be generated.

Performing model algebra

By changing the contents of the identifier sets that you have entered at the `Variables` and `Constraints` attributes of a mathematical program you can perform a simple form of *model algebra*. That is, you can investigate the effects of adding or removing constraints from within the graphical interface. Furthermore, it allows you to reconfigure your model based on the value of your model data.

Synchronizing variable and constraint sets

When changing the contents of either the variable or the constraint set of a mathematical program, you may find that the contents of the other set also needs some adjustment. For instance, adding a variable to a mathematical program makes no sense if there are no constraints that refer to it. AIMMS offers two special set-valued functions to help you to accomplish this task.

The function `VariableConstraints`

The function `VariableConstraints` takes a subset of the predefined set `AllVariables` as its argument, and returns a subset of the predefined set `AllConstraints`. The resulting constraint set contains all constraints which use one or more of the variables in the argument set.

The function `ConstraintVariables`

The function `ConstraintVariables` performs the opposite task. It takes a subset of the set `AllConstraints` as its arguments, and returns a subset of the set `AllVariables`. The resulting variable set contains all variables which are referred to in one or more constraints in the argument set. Also included are all variables referred to in the definitions of other variables inside the set.

Example

Consider the use of the functions `VariableConstraints` and `ConstraintVariables` in conjunction with the following declaration of a mathematical program.

```
MathematicalProgram PartialTransportModel {
    Objective   : TransportCost;
    Direction   : minimize;
    Constraints : PartialConstraintSet;
    Variables   : PartialVariableSet;
}
```

Assume that the set PartialVariableSet contains a subset of the variables declared in the model. Further assume that you would like to build up the contents of the set PartialConstraintSet together with the required additions to PartialVariableSet so that the contents of both sets are maximal. This is referred to as their transitive closure. By successively calling the functions VariableConstraints and ConstraintVariables, the following loop computes the transitive closure of the variable and constraint sets.

```
repeat
   PreviousCardinality  := Card( PartialVariableSet );
   PartialConstraintSet := VariableConstraints( PartialVariableSet   );
   PartialVariableSet   := ConstraintVariables( PartialConstraintSet );

   break when Card( PartialVariableSet ) = PreviousCardinality;
endrepeat ;
```

The break occurs when the set PartialVariableSet has not increased in size.

The Type attribute

With the Type attribute of a mathematical program you can prescribe a solution type. When the specified type is not compatible with the generated mathematical program, AIMMS will return an error message. You can override the type during a SOLVE statement using a WHERE clause for the type option. You can use this, for instance, to easily switch between the mip and rmip types.

Available types

A complete list of the mathematical program types available within AIMMS is given in Table 15.2. Most are self-explanatory. When the type rmip is specified, all integer variables are treated as continuous within their bounds. The rmip type is the global version of the Relax attribute associated with individual variables (see also Section 14.1). The types ls and nls can only be selected in the absence of the Objective attribute.

The Convention attribute

You can use the Convention attribute to specify the unit convention that you want to be used for scaling the variables and constraints in your mathematical program. For further details on this issue you are referred to Section 32.8.

The ViolationPenalty attribute

With the ViolationPenalty attribute you can instruct AIMMS to automatically add artificial terms to the constraints of your mathematical program to help resolve and/or track infeasibilities in your mathematical program. Infeasibility analysis and the use of the ViolationPenalty attribute is discussed in full detail in Section 15.4.

Type	Description
`lp`	linear program
`ls`	linear system
`qp`	quadratic program
`nlp`	nonlinear program
`nls`	nonlinear system
`mip`	mixed integer program
`rmip`	relaxed mixed integer program
`minlp`	mixed integer nonlinear program
`rminlp`	relaxed mixed integer nonlinear program
`qp`	quadratic program
`miqp`	mixed integer quadratic program
`qcp`	quadratic constraint program
`miqcp`	mixed integer quadratic constraint program
`network`	pure network program
`mcp`	mixed complementarity program
`mpcc`	mathematical program with complementarity constraint

Table 15.2: Available model types with Aimms

15.2 Suffices and callbacks

Suffices

A mathematical program has a number of suffices which can be used for various purposes. Typical examples are:

- To obtain information about the solution process. This information is filled in by the solver at the end of the solution process. These suffixes are presented in Table 15.3.
- To determine when and how to activate a callback procedure. This information can be filled in between solution steps. See also Chapter 16 where an alternative method for callbacks is presented. These suffixes are presented in Table 15.4.
- To get statistics of the generated mathematical program. These statistics are determined when the generated mathematical program is constructed. These suffixes are presented in Table 15.5.

Solver callbacks

After each iteration the external solver calls back to the Aimms system to offer Aimms the opportunity to take control. Aimms, in turn, allows you to execute a procedure which is referred to as a *callback procedure*. Once the callback procedure has finished, the control is returned to the external solver to continue with the next iteration. By including a callback procedure you can perform several tasks such as:

- inspect the current status of the solution process,

Suffix	**Meaning**
`Objective`	Current objective value
`LinearObjective`	Current linear objective value
`Incumbent`	Current incumbent value
`BestBound`	Best LP solution
`ProgramStatus`	Current program status
`SolverStatus`	Current solver status
`Iterations`	Current number of iterations
`Nodes`	Current number of nodes (`mip`, `miqp`, and `miqcp` only)
`GenTime`	Current generation time in [second]
`SolutionTime`	Current solution time in [second]
`NumberOfBranches`	Number of nodes visited by a CP solver
`NumberOfFails`	Number of leaf nodes without solution in a CP search tree
`NumberOfInfeasibilities`	Final number of infeasibilities
`SumOfInfeasibilities`	Final sum of the infeasibilities

Table 15.3: Suffices of a mathematical program filled by the solver

- update one or more model parameters, which can be used, for instance, to provide a graphical overview of the solution process,
- retrieve (part of) the current solution, and
- abort the solution process, and

You can nominate any procedure as a callback procedure by assigning its name to the suffix `CallbackProcedure` of the associated mathematical program as in:

```
TransportModel.CallbackProcedure := 'MyCallbackProcedure' ;
```

Note that values assigned to the suffix `CallbackProcedure` or any of the other suffices holding the name of a callback procedure, must be elements of the predefined set `AllProcedures`. Therefore, if you assign a literal procedure name to such a suffix, you should make sure to quote it, as illustrated in the example above.

When activated

Callback procedures under your control may cause a considerable computational overhead, and should only be activated when necessary. To give you control of the frequency of callbacks, AIMMS provide three separate suffices to trigger a callback procedure. Specifically, a callback procedure can be called

- after a specified number of iterations,
- after a specified number of seconds,
- after a change of status of the solution process, or
- at every new incumbent value during the solution process of a mixed integer program.

Suffix	**Meaning**
`CallbackProcedure`	Name of callback procedure
`CallbackIterations`	Return to callback after this number of iterations
`CallbackTime`	Name of callback procedure to be called after some elapsed time
`CallbackStatusChange`	Name of callback procedure to be called after a status change
`CallbackNewIncumbent`	Name of callback procedure to be called for every new incumbent value
`CallbackAddCut`	Name of callback procedure to be called to add additional cuts (Cplex and Gurobi)
`CallbackReturnStatus`	Return status of callback
`CallbackAOA`	Name of AOA callback procedure

Table 15.4: Suffices of a mathematical program stated by the user

Activated after iterations

With the suffix `CallbackIterations` you can indicate after how many iterations the callback procedure specified by the `CallbackProcedure` suffix must be called again. If you specify the number 0 (default), no such callbacks will be made.

Activated after time

With the suffix `CallbackTime` you specify the name of the callback procedure to be called when a certain number of seconds has elapsed. When not specified (the default), no such callbacks are made.

Activated after status change

With the suffix `CallbackStatusChange` you specify the name of the callback procedure to be performed when the status of the solution process changes. When not specified (the default), no such callbacks are made.

Activated after new incumbent

With the suffix `CallbackNewIncumbent` you specify the name of the callback procedure to be performed when the solver finds a new incumbent value during the solution process of a mixed integer program. When not specified (the default), no such callbacks are made.

Watch objective values

During a callback procedure you can access various objective values as they are reported by the solver during a mixed integer program through several suffices of the mathematical program at hand. The following suffices provide information about the objective values:

- through the suffix `Incumbent` you can obtain the objective value of the best integer solution found so far,
- through the suffix `LinearObjective` you can obtain the objective value of the relaxation of the current node in the branch-and-bound process being investigated, and

Suffix	Meaning
`SolverCalls`	Total number of applied `SOLVE`'s
`NumberOfConstraints`	Number of individual constraints
`NumberOfVariables`	Number of individual variables
`NumberOfNonzeros`	Number of nonzeros
`NumberOfIntegerVariables`	Number of individual integer variables
`NumberOfIndicatorConstraints`	Number of individual constraints with an activating condition
`NumberOfSOS1Constraints`	Number of individual SOS1 constraints
`NumberOfSOS2Constraints`	Number of individual SOS2 constraints
`NumberOfNonlinearConstraints`	Number of individual nonlinear constraints
`NumberOfNonlinearVariables`	Number of individual nonlinear variables
`NumberOfNonlinearNonzeros`	Number of nonlinear nonzeros

Table 15.5: Suffices of a mathematical program statistics from Aimms

- through the suffix `Objective` you can obtain the current objective value reported by the solver at the precise time of the callback.

For mixed integer programs the suffix `Objective` will be meaningless in most cases during the solution process.

Watch intermediate solution values

In a callback procedure you can access the current solution values of the variables in the mathematical program, and assign these to other identifiers in your model. One possible use of this feature is to store multiple feasible integer solutions of a mixed integer linear program.

The procedure `RetrieveCurrentVariableValues`

For some solvers there may be a considerable overhead involved to retrieve the current variable values during the running solution process. Therefore, Aimms will only do so when you explicitly call the procedure

`RetrieveCurrentVariableValues(`*VariableSet*`)`

With the *VariableSet* argument you can specify the subset of the set `AllVariables` consisting of all (symbolic) variables for which you want the current values to be retrieved. When you call this procedure outside the context of a solver callback procedure, Aimms will produce a runtime error.

Adding additional cuts

When you want to add additional cuts during the solution process of a mixed integer program, you should install a callback procedure to generate these constraints using the `CallbackAddCut` suffix. This procedure is called at every node that has an LP-optimal solution with an objective function value below the current cutoff and is integer infeasible. The procedure allows you to add individual constraints using the `GenerateCut(`*`row`*`, `*`local`*`)` function. The *row* argument should always be a scalar reference to an existing constraint name in your model. The *local* argument should be a scalar binary that indicates whether the cut is a local cut (value 1) or a global one (value 0). The *local* argument is an optional argument, and has a default of 1.

Example

Consider a model with the following constraint.

```
Constraint Triangle_Cut {
    IndexDomain  : (i1,i2,i3) | (i1 < i2) and (i2 < i3);
    Definition   : x(i1) + x(i2) + x(i3) - y(i1,i2) - y(i1,i3) - y(i2,i3) <= 1;
}
```

Then the following piece of code, when specified as the procedure body of the `CallbackAddCut` procedure, will only add those triangle cuts that are violated.

```
    RetrieveCurrentVariableValues(AllVariables);

    for ( (i1,i2,i3) | (i1 < i2) and (i2 < i3) ) do
        if ( x(i1) + x(i2) + x(i3) - y(i1,i2) - y(i1,i3) - y(i2,i3) > 1 + eps ) then
            GenerateCut( Triangle_Cut(i1,i2,i3), 1 );
        endif;
    endfor;
```

Aborting the solution process

When you want to abort the solution process, you can set the suffix `Callback-ReturnStatus` to `'abort'` during the execution of your callback procedure, as in:

```
     TransportModel.CallbackReturnStatus := 'abort' ;
```

After aborting the process, AIMMS will retrieve the current solution and set the final solver status to `UserInterrupt`.

Example

Consider a mathematical program `TransportModel` which incorporates a callback procedure. The following callback procedure will abort the solution process if the total solution time exceeded 1800 seconds, and if the progress is less than 1% compared to the last nonzero objective function value.

```
    if ( TransportModel.SolutionTime > 1800 [second] and PreviousObjective and
         (TransportModel.Objective - PreviousObjective) < 0.01*PreviousObjective )
    then
       TransportModel.CallbackReturnStatus := 'abort';
    else
       PreviousObjective := TransportModel.Objective;
    endif;
```

Solver and program status

Both the `ProgramStatus` and the `SolverStatus` suffix take their value in the predefined set `AllSolutionStates` presented in Table 15.6.

Program status	**Solver status**
`ProgramNotSolved`	`SolverNotCalled`
`Optimal`	`NormalCompletion`
`LocallyOptimal`	`IterationInterrupt`
`Unbounded`	`ResourceInterrupt`
`Infeasible`	`TerminatedBySolver`
`LocallyInfeasible`	`EvaluationErrorLimit`
`IntermediateInfeasible`	`Unknown`
`IntermediateNonOptimal`	`UserInterrupt`
`IntegerSolution`	`PreprocessorError`
`IntermediateNonInteger`	`SetupFailure`
`IntegerInfeasible`	`SolverFailure`
`InfeasibleOrUnbounded`	`InternalSolverError`
`UnknownError`	`PostProcessorError`
`NoSolution`	`SystemFailure`

Table 15.6: Mathematical program and solver status

15.3 The `SOLVE` statement

The `SOLVE` statement

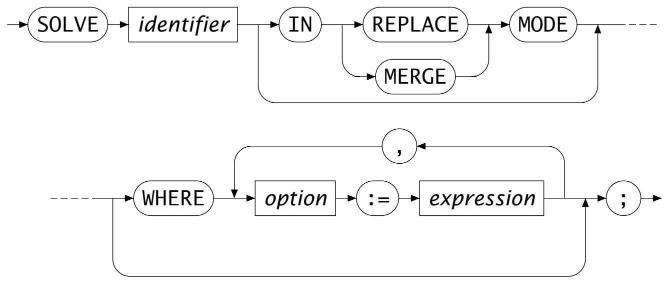
With the `SOLVE` statement you can instruct Aimms to compute the solution of a `MathematicalProgram`, resulting in the following actions.

- Aimms determines which solution method(s) are appropriate, and checks whether the specified type is also appropriate.
- Aimms then generates the Jacobian matrix (first derivatives of all the constraints), the bounds on all variables and constraints, and an objective where appropriate.
- Aimms communicates the problem to an underlying solver that is able to perform the chosen solution method.
- Aimms finally reads the computed solution back from the solver.

Syntax

In addition to initiating the solution process of a `MathematicalProgram`, you can also use the `SOLVE` statement to provide local overrides of particular Aimms settings that influence the way in which the solution process takes place. The syntax of the `SOLVE` statement follows.

solve-statement :



Replace and merge mode

You can instruct AIMMS to read back the solution in either *replace* or *merge* mode. If you do not specify a mode, AIMMS assumes replace mode. In replace mode AIMMS will, before reading back the solution of the mathematical program, remove the values of the variables in the `Variables` set of the mathematical program for all index tuples except those that are fixed

- because they are not within their current domain (i.e. inactive),
- through the `NonvarStatus` attribute or the `.NonVar` suffix of the variable,
- because they are outside the planning interval of a `Horizon` (see Section 33.3), or
- because their upper and lower bounds are equal.

In merge mode AIMMS will only replace the *individual variable values* involved in the mathematical program. This mode is very useful, for instance, when you are iteratively solving subproblems which correspond to slices of the symbolic variables in your model.

Infeasible and unbounded problems

Whenever the invoked solver finds that a mathematical program is infeasible or unbounded, AIMMS will assign one of the special values `na`, `inf` or `-inf` to the objective variable. For you, this will serve as a reminder of the fact that there is a problem even when you do not check the `ProgramStatus` and `SolverStatus` suffices. For all other variables, AIMMS will read back the last values computed by the solver just before returning with infeasibility or unboundedness.

Temporary option settings

Sometimes you may need some temporary option settings during a single `SOLVE` statement. Instead of having to change the relevant options using the `OPTION` statement and set them back afterwards, AIMMS also allows you to specify values for options that are used only during the current `SOLVE` statement. The syntax is similar to that of the `OPTION` statement.

Also for attributes

Apart from specifying temporary option settings you can also use the `WHERE` clause to override the `type` and `direction` attributes specified in the declaration of the mathematical program, as well as the `solver` to use for the solution process.

Example

The following SOLVE statement selects 'cplex' as its solver, sets the model type to 'rmip', and sets the CPLEX option LpMethod to 'Barrier'.

```
solve TransportModel in replace mode
    where solver    := 'cplex',
          type      := 'rmip',
          LpMethod  := 'Barrier' ;
```

15.4 Infeasibility analysis

Infeasibility analysis

One of the more daunting tasks in mathematical programming is to find the cause of an infeasible mathematical program. Such infeasibilities may occur

- either when you are developing a new model due to modeling errors,
- or in a complete (and well-tested), model-based, end-user application employed by your customers due to inconsistencies in the model data.

Infeasibilities due to modeling errors

There are several types of modeling errors that you can make during the development of a mathematical program that can lead to hard-to-explain infeasibilities. The most common are:

- simple typing errors, leading, for instance, to a wrong variable being referenced in a constraint,
- a logical flaw in the model formulation, i.e. the formulation of one or more constraints just makes no sense,
- the domain restriction of a constraint is not restrictive enough, i.e. constraints are generated that should not be generated,
- the domain restriction of a variable is wrong, leading to too many or too few terms being generated in constraints referring to such a variable, or
- the restriction in iterative operators (such as SUM or PROD) in the definition of constraints or defined variables is wrong, leading to too many or too little terms being generated in that particular constraint.

In general, trying to find infeasibilities that occur during model development may force you to generate a constraint listing of your mathematical program and carefully examine the generated constraints in order to find the modeling error.

Infeasibilities due to data inconsistencies

Even when the formulation of a mathematical program is internally consistent, and shipped as an end-user application to your customers, infeasibilities may occur due to inconsistencies in the model data. The most common data errors are:

- inconsistencies in the structural data defining the topology of a model, e.g. in a network model a demand node may have been added for which no incoming arcs have been specified, or

- inconsistencies in the quantitive model data, e.g. to total demand exceeds the total supply.

While most data inconsistencies may be detected by methodically checking the consistency all input data prior to actually solving the mathematical program (for example, by using `Assertions`, see also Section 25.2), it is often hard to cover all possible data inconsistencies.

Adding excess variables

A commonly used approach to try and deal with infeasibilities, is to add explicit excess variables to all or some constraints in a model, along with a penalty term in the objective that will keep all excess variables equal to 0 if the model is feasible. If this procedure is executed properly, the *modified* mathematical program will always be feasible, while the *original* mathematical program is feasible if and only if the excess variables are all equal to 0. In the case of an infeasibility, an examination of the excess variables may provide useful information about the cause of infeasibility.

Laborious procedure

While adding excess variables to your model may certainly help you to resolve any infeasibilities, the process of manually adding these excess variables to a mathematical program is laborious and error-prone:

- you have to add the declarations of the excess variables for all (or some) constraints in your model,
- the selected constraints have to be modified to include these excess variables, and
- the objective has to be modified to include the excess-related penalty terms.

In addition, adding excess variables may considerably increase the size of the generated matrix, so you may want to write supporting code to exclude the excess variables from your mathematical program unless you encounter an infeasibility.

15.4.1 Adding infeasibility analysis to your model

The `Violation-Penalty` attribute

To ease the manual process described above, AIMMS offers support to *automatically* extend your mathematical program with excess variables during the generation of the matrix for the solver. You enable this feature through the `ViolationPenalty` attribute of a `MathematicalProgram` declaration. The value of the `ViolationPenalty` attribute must be either a

- 1-dimensional parameter with index domain `AllVariablesConstraints`, or
- 2-dimensional parameter defined over `AllVariablesConstraints` and `All-ViolationTypes`.

The predefined set `AllVariablesConstraints` is a subset of the set `AllIdenti-fiers` and contains the names of all the variables and constraints in your model. Through one of these two types of parameters you can specify for which variables and constraints in your mathematical program AIMMS must generate excess variables, as well as the penalty coefficient of these excess variables in the modified objective.

The set `AllViolationTypes`

The predefined set `AllViolationTypes` is a fixed set containing the three types of possible violations for which AIMMS can generate excess variables. The elements in the set `AllViolationTypes` are

- `Lower`: generate excess variables for the violation of a lower bound,
- `Upper`: generate excess variables for the violation of an upper bound, and
- `Definition`: generate excess variables for the violation of the equality between a defined variable and its definition.

Interpretation of `ViolationPenalty` attribute

If a parameter you entered in the `ViolationPenalty` attribute contains no data, AIMMS will generate the mathematical program without any generated excess variables. If you specify a 2-dimensional parameter which is not empty, all values must be nonnegative or assume the special value `ZERO` (see also Section 6.1.1), and AIMMS will interpret its contents as follows.

Penalty for objective variable

The modified objective will include the original objective, unless a value of `ZERO` has been assigned to *any* of the `Lower`, `Upper` or `Definition` violation types for the original objective variable. AIMMS will disregard any other penalty value than `ZERO` assigned to the objective variable. Note that by including the original objective the penalized mathematical program may become unbounded.

Penalty for constraints

AIMMS will add nonnegative excess variables for the violation of a (finite) lower and/or upper bound of every constraint for which a penalty value other than 0.0 has been specified for the `Lower` and/or `Upper` violation type, respectively. If a bound is infinite, no corresponding excess variable will be generated. A penalty term will be added to the modified objective consisting of the product of the specified (nonnegative) penalty coefficient times the excess variable associated with the constraint, unless a penalty of `ZERO` has been specified in which case the corresponding term will not be added to the modified objective.

Penalty for variables

AIMMS will add nonnegative excess variables for the violation of a (finite) lower and/or upper bound of every variable for which a penalty value other than 0.0 has been specified for the `Lower` and/or `Upper` violation type, respectively. If a bound is infinite, no corresponding excess variable will be generated. A penalty term will be added to the modified objective consisting of the product of the specified (nonnegative) penalty coefficient times the excess variable associated with the variable, unless a penalty of `ZERO` has been specified in which case the corresponding term will not be added to the modified objective. The effect of using `Lower` and/or `Upper` violations is that the variable can assume values outside their bounds throughout the mathematical program.

Penalty for variable definitions

Aimms will add nonnegative excess variables for the violation of the equality between a defined variable and its definition for every defined variable for which a penalty value other than 0.0 has been specified for the `Definition` violation type. A penalty term will be added to the modified objective consisting of the product of the specified (nonnegative) penalty times the excess variable(s) associated with the constraint expressing the equality, unless a penalty of `ZERO` has been specified in which case the corresponding term(s) will not be added to the modified objective.

Definition versus lower/upper violations

You can both use the `Lower` and/or `Upper` violation types and `Definition` violation type to compensate for a violation between the value of the defined variable and its definition. However, when you use the `Definition` violation type, the value of the variable will remain within its specified bounds throughout the mathematical program. It is up to you to decide which violation type suits your needs best for a particular defined variable.

Interpretation of 1-dimensional parameter

If you specify a 1-dimensional parameter for the `ViolationPenalty` attribute, Aimms will interpret this parameter as if it were a 2-dimensional parameter, with the same value for all three violation types `Lower`, `Upper` and `Definition`.

15.4.2 Inspecting your model for infeasibilities

Finding violations

After you have let Aimms extend your model with excess variables to find an infeasibility, you must inspect the variables and constraints in your model to find the violations. Aimms allows you to do this through the use of two suffices, the `.Violation` suffix and the `.DefinitionViolation` suffix.

The `.Violation` suffix...

The `.Violation` suffix denotes the amount by which a variable or constraint violates its lower or upper bound. If you have specified a nonzero violation penalty for the `Upper` violation type, the `.Violation` suffix can assume positive values, while it can assume negative values whenever you have specified a nonzero violation penalty for the `Lower` violation type.

... for variables

For variables the `.Violation` suffix denotes the amount by which the variable violates its

- upper bound (if the suffix assumes a positive value), or
- lower bound (if the suffix assumes a negative value).

... for constraints

For constraints the `.Violation` suffix denotes the amount by which the constraint violates its

- upper bound (if the suffix assumes a positive value),
- lower bound (if the suffix assumes a negative value, for ranged constraints).

If the constraint is an equality constraint, the `.Violation` suffix denotes the (positive or negative) amount by which the left hand side differs from the (constant) right hand side.

The `.DefinitionViolation` suffix

With the `.DefinitionViolation` suffix, you can locate violations in the definitions of defined variables for which you have specified a positive penalty for the `Definition` violation type. The value of the suffix denotes the (positive or negative) amount by which the defined variable differs from its definition. Note that a defined variable may violate both its bounds and its definition, depending on the type of allowed violations you have specified.

Locating violations

To locate violations in a model which was extended by AIMMS with excess variables, you may use the `Card` function to locate variables and constraints with nonzero `.Violations` suffices. The following example shows how to proceed, where `v` is assumed to be an index in `AllVariables`.

```
for ( v | Card(v, 'Violation'}) ) do

    ! Take any action that you want to perform on this violated variable

endfor;
```

15.4.3 Application to goal programming

Goal programming ...

In goal programming a distinction is made between *hard constraints* that cannot be violated and *soft constraints*, which represent goals or targets one would like to achieve. The objective function in goal programming is to minimize the weighted sum of deviations from the goals set by the soft constraints.

... interpreted as violations

In AIMMS, goal programming can be easily implemented using the `ViolationPenalty` attribute of a mathematical program, without the need to modify the formulation of all soft constraints. For each soft constraint in your goal programming model, you can assign the appropriate weight to the `ViolationPenalty` attribute to penalize deviations from the set target for that constraint.

Inspecting deviations

Through the `.Violation` suffix of constraints and variables you can inspect the deviations from the goals of the soft constraints in your goal programming model.