AIMMS Language Reference - Variable and Constraints Declaration

AIMMS 4

AIMMS B.V.
Diakenhuisweg 29-35
2033 AP Haarlem
The Netherlands
Tel.: +31 23 5511512

AIMMS Inc.
11711 SE 8th Street
Suite 303
Bellevue, WA 98005
USA
Tel.: +1 425 458 4024

AIMMS Pte. Ltd.
55 Market Street #10-00
Singapore 048941
Tel.: +65 6521 2827

AIMMS
SOHO Fuxing Plaza No.388
Building D-71, Level 3
Madang Road, Huangpu District
Shanghai 200025
China
Tel.: ++86 21 5309 8733

Email: info@aimms.com
WWW: www.aimms.com

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

Part V

Optimization Modeling Components

Chapter 14

Variable and Constraint Declaration

Terminology

The word variable does not have a uniform meaning. In general, programmers view a variable as a known but varying quantity that receives its value through direct assignments. *However, in the context of constraints in* AIMMS*, the word variable denotes an unknown quantity.* Constraints can be grouped together to form a system of simultaneous equations and/or inequalities, which is referred to as a *mathematical program*. Variables in a mathematical program are assigned values when a solver (a solution algorithm) finds a solution for the unknowns in the system.

Similarity of parameters and variables

When used outside the scope of constraints and the solution of mathematical programs, variables in AIMMS behave essentially the same as parameters in AIMMS. Like parameters, variables can be initialized, used as known quantities in assignment statements, and be referred to as data from within the graphical user interface.

14.1 `Variable` declaration and attributes

Declaration and attributes

Variables have some additional attributes above those of parameters. These extra attributes are used to steer a solver, or to hold additional information about solution values provided by the solver. The possible attributes of variables are given in Table 14.1.

Index domain for variables

By specifying the `IndexDomain` attribute you can restrict the domain of a variable in the same way as that of a parameter. For variables, however, the domain restriction has an additional effect. During the generation of individual constraints AIMMS will reduce the size of the generated mathematical program by including only those variables that satisfy all domain restrictions.

The `Range` attribute

The values of the `Range` attribute of variables determine the bounds that are passed on to the solver. In addition, during an assignment, the `Range` attribute restricts the range of allowed values that can be assigned to a particular interval (as for parameters). The possible values for the `Range` attribute are:

Attribute	Value-type	See also page
`IndexDomain`	*index-domain*	44
`Range`	*range*	45
`Default`	*constant-expression*	46
`Unit`	*unit-expression*	47, 508
`Priority`	*expression*	
`NonvarStatus`	*expression*	
`RelaxStatus`	*expression*	
`Property`	`NoSave`, *numeric-storage-property*, `Inline` `SemiContinuous`, `Basic`, `Stochastic`, `Adjustable` `ReducedCost`, `ValueRange`, `CoefficientRange`, *constraint-related-sensitivity-property*	34, 47
`Text`	*string*	19
`Comment`	*comment string*	19
`Definition`	*expression*	34, 47
`Stage`	*expression*	220, 312
`Dependency`	*expression*	220, 339

Table 14.1: `Variable` attributes

- one of the predefined ranges `Real`, `Nonnegative`, `Nonpositive`, `Integer` or `Binary`,
- any one of the interval expressions $[a,b]$, $[a,b)$, $(a,b]$ or (a,b), where a and b can be a constant number, `inf`, `-inf`, or a parameter reference involving some or all of the indices on the index list of the declared variable,
- any enumerated integer set expression, e.g. `{`a `..` b`}` with a and b as above, or
- an integer set identifier.

If you specify `Real`, `Nonnegative`, `Nonpositive`, or an interval expression, AIMMS will interpret the variable as a continuous variable. If you specify `Integer`, `Binary` or an integer set expression, AIMMS will interpret the variable as a binary or integer variable.

Example

The following example illustrates a simple variable declaration.

```
Variable Transport {
    IndexDomain  : (i,j) in Connections;
    Range        : [ MinTransport(i), Capacity(i,j) ];
}
```

The declaration of the variable `Transport(i,j)` sets its lower bound equal to `MinTransport(i)` and its upper bound to `Capacity(i,j)`. When generating the mathematical program, the variable `Transport` will only be generated for those tuples (i,j) that lie in the set Connections. Note that the specification of the lower bound only uses a subdomain (i) of the full index domain of the variable (i,j).

The .Lower and .Upper suffices

Besides using the Range attribute to specify the lower and upper bounds, you can also use the .Lower and .Upper suffices in assignment statements to accomplish this task. The .Lower and .Upper suffices are attached to the name of the variable, and, as a result, the corresponding bounds are defined for the entire index domain. This may lead to increased memory usage when variables share their bounds for slices of the domain. For this reason, you are advised to use the Range attribute as much as possible when specifying the lower and upper bounds.

When allowed

You can only make a bound assignment with either the .Lower or .Upper suffix when you have not used a parameter reference (or a non-constant expression) at the corresponding position in the Range attribute. Bound assignments via the .Lower and .Upper suffices must always lie within the range specified in the Range attribute.

Example

Consider the variable Transport declared in the previous example. The following assignment to Transport.Lower(i,j) is not allowed, because you have already specified a parameter reference at the corresponding position in the Range attribute.

```
    Transport.Lower(i,j) := MinTransport(i) ;
```

On the other hand, given the following declaration,

```
Variable Shipment {
    IndexDomain  : (i,j) in Connections;
    Range        : Nonnegative;
}
```

the following assignment is allowed:

```
    Shipment.Lower(i,j) := MinTransport(i);
```

Aimms will produce a run-time error message if any value of MinTransport(i) is less than zero, because this violates the bound in the Range attribute of the variable Shipment.

The Default attribute

Variables that have not been initialized, evaluate to a default value automatically. These default values are also passed as initial values to the solver. You can specify the default value using the Default attribute. The value of this attribute *must* be a constant expression. If you do not provide a default value, Aimms will use a default of 0.

The `Unit` attribute

Providing a `Unit` for every variable and constraint in your model will help you in a number of ways.

- Aimms will help you to check the consistency of all your constraints and assignments in your model, and
- Aimms will use the units to scale the model that is sent to the solver.

Proper scaling of a model will generally result in a more accurate and robust solution process. You can find more information on the definition and use of units to scale mathematical programs in Chapter 32.

The `Definition` attribute

It is not unusual that symbolic constraints in a model are equalities defining just one variable in terms of others. Under these conditions, it is preferable to provide the definition of the variable through its `Definition` attribute. As a result, you no longer need to specify extra constraints for just variable definitions. In the constraint listing, the constraints associated with a defined variable will be listed with a generated name consisting of the name of the variable with an additional suffix "`_definition`".

Example

The following example defines the total cost of transport, based on unit transport cost and actual transport taking place.

```
Variable TransportCost {
    Definition : sum( (i,j), UnitTransportCost(i,j)*Transport(i,j) );
}
```

14.1.1 The `Priority`, `Nonvar` and `RelaxStatus` attributes

The `Priority` attribute

With the `Priority` attribute you can assign priorities to integer variables (or continuous variables when using the solver Baron). The value of this attribute must be an expression using some or all of the indices in the index domain of the variable, and must be nonnegative and integer. All variables with priority zero will be considered last by the branch-and-bound process of the solver. For variables with a positive priority value, those with the highest priority value will be considered first.

The `.Priority` suffix

Alternatively, you can specify priorities through assignments to the `.Priority` suffix. This is only allowed if you have not specified the `Priority` attribute. In both cases, you can use the `.Priority` suffix to refer to the priority of a variable in expressions.

Use of priorities

The solution algorithm (i.e. solver) for integer and mixed-integer programs initially solves without the integer restriction, and then adds this restriction one variable at a time according to their priority. By default, all integer variables have equal priority. Some decisions, however, have a natural order in time or space. For example, the decision to build a factory at some site comes before the decision to purchase production capacity for that factory. Obeying this order naturally limits the number of subsequent choices, and could speed up the overall search by the solution algorithm.

The `NonvarStatus` attribute

You can use the `NonvarStatus` attribute to tell AIMMS which variables should be considered as parameters during the execution of a `SOLVE` statement. The value of the `NonvarStatus` attribute must be an expression in some or all of the indices in the index list of the variable, allowing you to change the nonvariable status of individual elements or groups of elements at once.

Positive versus negative values

The sign of the `NonvarStatus` value determines whether and how the variable is passed on to the solver. The following rules apply.

- If the value is 0 (the default value), the corresponding individual variable is generated, along with its specified lower and upper bounds.
- If the value is negative, the corresponding individual variable is still generated, but its lower and upper bounds are set equal to the current value of the variable.
- If the value is positive, the corresponding individual variable is no longer generated but passed as a constant to the solver.

When you specify a negative value, you will still be able to inspect the corresponding reduced cost values. In addition, you can modify the nonvariable status to zero without causing AIMMS to regenerate the model. When you specify a positive value, the size of the mathematical program is kept to a minimum, but any subsequent changes to the nonvariable status will require regeneration of the model constraints.

The `.NonVar` suffix

Alternatively, you can change the nonvariable status through assignments to the `.NonVar` suffix. This is only allowed if you have not specified the `NonvarStatus` attribute. In both cases, you can use the `.NonVar` suffix to refer to the variable status of a variable in expressions.

When to change the nonvariable status

By altering the nonvariable status of variables you are essentially reconfiguring your mathematical program. You could, for instance, reverse the role of an input parameter (declared as a variable with negative nonvariable status) and an output variable in your model to observe what input level is required to obtain a desired output level. Another example of temporary reconfiguration is to solve a smaller version of a mathematical program by first discarding selected variables, and then changing their status back to solve the larger mathematical program using the previous solution as a starting point.

The RelaxStatus attribute

With the RelaxStatus attribute you can tell AIMMS to relax the integer restriction for those tuples in the domain of an integer variable for which the value of the relax status is nonzero. AIMMS will generate continuous variables for such tuples instead, i.e. variables which may assume any real value between their bounds.

The .Relax suffix

Alternatively, you can relax integer variables by making assignments to the .Relax suffix. This is only allowed if you have not specified the RelaxStatus attribute. In both cases, you can use the .Relax suffix to refer to the relax status of a variable in expressions.

When to relax variables

When solving large mixed integer programs, the solution times may become unacceptably high with an increase in the number of integer variables. You can try to resolve this by relaxing the integer condition of some of the integer variables. For instance, in a multi-period planning model, an accurate integer solution for the first few periods and an approximating continuous solution for the remaining periods may very well be acceptable, and at the same time reduce solution times drastically.

Effect on mathematical program type

As you will see in Chapter 15, there are several types of mathematical programs. By changing the nonvariable and/or relax status of variables you may alter the type of your mathematical program. For instance, if your constraints contains a nonlinear term x*y, then changing the nonvariable status of either x or y will change it into a linear term. Eventually, this may result in a nonlinear mathematical program becoming a linear one. Similarly, changing the nonvariable or relax status of integer variables may at some point change a mixed integer program into a linear program.

14.1.2 Variable properties

Properties of variables

Variables can have one or more of the following properties: NoSave, Inline, SemiContinuous, ReducedCost, CoefficientRange, ValueRange, Stochastic, and Adjustable. They are described in the paragraphs below.

Use of PROPERTY statement

You can also change the properties of a variable during the execution of your model by calling the PROPERTY statement. Identifier properties are changed by adding the property name as a suffix to the identifier name in a PROPERTY statement. When the value is set to off, the property no longer holds.

The NoSave property

With the property NoSave you indicate that you do not want to store data associated with this variable in a case. This property is especially suited for those identifiers that are intermediate quantities in the model, and that are not used anywhere in the graphical end-user interface.

Inline variables

With the property `Inline` you can indicate that AIMMS should substitute all references to the variable at hand by its defining expression when generating the constraints of a mathematical program. Setting this property only makes sense for defined variables, and will result in a mathematical program with less rows and columns but with a (possibly) larger number of nonzeros. After the mathematical program has been solved, AIMMS will compute the level values of all inline variables by evaluating their definition. However, no sensitivity information will be available.

Semi-continuous variables

To any continuous or integer variable you can assign the property `SemiContinuous`. This indicates to the solver that this variable is either zero, or lies within its specified range. Not all solvers support semi-continuous variables. In the latter case, AIMMS will automatically add the necessary constraints to the model.

14.1.3 Sensitivity related properties

Basic, superbasic, and nonbasic variables

With the `Basic` property you can instruct AIMMS to retrieve basic information of a specific variable from the solver. If retrieved, basic information can be accessed through the `.Basic` suffix. The basic information is presented as an element in the predefined AIMMS set `AllBasicValues` (i.e. {*Basic, Nonbasic, Superbasic*}). In linear programming a variable will either be basic or nonbasic, while in nonlinear programming the number of variables with zero reduced cost can be larger than the number of constraints. The solution algorithm then divides these variables into so-called *basics* and *superbasics*. The basic variables define a square system of nonlinear equations which is solved for fixed values of the remaining variables. The superbasics are assigned a fixed value between their bounds, while the nonbasics take their value at a bound.

The `ReducedCost` property

You can use the `ReducedCost` property to specify whether you are interested in the reduced cost values of the variable after each `SOLVE` step. Storing the reduced costs of all variables may be very memory consuming, therefore, the default in AIMMS is not to store these values. If reduced costs are requested, the stored values can be accessed through the suffices `.ReducedCost` or `.m`.

Interpretation of reduced cost

The reduced cost indicates by how much the cost coefficient in the objective function should be reduced before the variable becomes active (off its bound). By definition, the reduced cost value of a variable between its bounds is zero. The precise mathematical interpretation of reduced cost is discussed in most text books on mathematical programming. Note: if a basic or superbasic variable has a reduced cost of zero then it will be displayed as 0.0, but if a nonbasic variable has a reduced cost of zero then it will be displayed as `ZERO`.

Unit of reduced cost

When the variables in your model have an associated unit (see Chapter 32), special care is required in interpreting the values returned through the `.ReducedCost` suffix. To obtain the reduced cost in terms of the units specified in the model, the values of the `.ReducedCost` suffix must be scaled as explained in Section 32.5.1.

The property `CoefficientRange`

With the property `CoefficientRange` you request AIMMS to conduct a first type of sensitivity analysis on this variable during a `SOLVE` statement of a linear program. The result of this sensitivity analysis are three parameters, representing the smallest, nominal, and largest values for the *objective coefficient* of the variable so that the optimal basis remains constant. Their values are accessible through the suffices `.SmallestCoefficient`, `.NominalCoefficient` and `.LargestCoefficient`.

The property `ValueRange`

With the property `ValueRange` you request AIMMS to conduct a second type of sensitivity analysis during a `SOLVE` statement of a linear program. The result of the sensitivity analysis are two parameters, representing the smallest and largest values that the *variable* can take while holding the objective value constant. Their values are accessible through the `.SmallestValue` and `.LargestValue` suffices.

Linear programs only

AIMMS only supports the sensitivity analysis conducted through the properties `CoefficientRange` and `ValueRange` for linear mathematical programs. If you want to apply these types of analysis to the final solution of a mixed-integer program, you should fix all integer variables to their final solution (using the `.NonVar` suffix) and re-solve the resulting mathematical program as a linear program (e.g. by adding the clause `WHERE type:='lp'` to the `SOLVE` statement).

Storage and computational costs

Setting any of the properties `ReducedCost`, `CoefficientRange` or `ValueRange` may result in an increase of the memory usage. In addition, the computations required to compute the `ValueRange` may considerably increase the total solution time of your mathematical program.

Constraint related properties

Whenever a defined variable (which is not declared `Inline`) is part of a mathematical program, AIMMS implicitly adds a constraint to the generated model expressing this definition. In addition to the variable-related sensitivity properties discussed in this section, you can specify the constraint-related sensitivity properties `ShadowPrice`, `RightHandSideRange` and `ShadowPriceRange` (see also Section 14.2) for such variables to obtain the sensitivity information that can be related to these constraint. You can access the requested sensitivity information by appending the associated suffices to the name of the defined variable.

14.1.4 Uncertainty related properties and attributes

Stochastic programming and robust optimization

The AIMMS modeling language offers facilities for both stochastic programs and robust optimization models. For both types of models you can specify special `Variable` properties and attributes to define uncertainty-related relationships.

The `Stochastic` property

Through the `Stochastic` property you can indicate that, within a stochastic model, the variable can hold scenario-dependent solutions. AIMMS will add a `Stage` attribute for every variable for which the `Stochastic` property has been set.

The `Stage` attribute

The value of the `Stage` attribute must be a numerical expression evaluating to in integer number indicating the stage at the end of which the variable takes its value during the solution process of a stochastic model. Stochastic programming, and the `Stochastic` property and `Stage` attribute are discussed in full detail in Section 19.2.

The `Adjustable` property

By setting the `Adjustable` property for a variable, you indicate that a variable in a robust optimization model has a functional dependency on some or all of the uncertain parameters in the model. If you declare a variable to be adjustable, the `Dependency` attribute also becomes available for that variable.

The `Dependency` attribute

Through the `Dependency` attribute you specify the precise collection of uncertain parameters on which the variable at hand depends. At this moment, AIMMS only supports affine relations between uncertain parameters and adjustable variables. The precise semantics of the `Dependency` attribute is discussed in Section 20.4.

14.2 `Constraint` declaration and attributes

Definitions

Constraints form the major mechanism for specifying a mathematical program in AIMMS. They are used to restrict the values of variables with interlocking relationships. *Constraints* are numerical relations containing expressions in terms of variables, parameters and constants.

Constraint attributes

The possible attributes of constraints are given in Table 14.2.

Domain restriction for constraints

Restricting the domain of constraints through the `IndexDomain` attribute influences the matrix generation process. Constraints are generated only for those tuples in the index domain that satisfy the domain restriction.

Attribute	Value-type	See also page
IndexDomain	*index-domain*	44
Unit	*unit-valued expression*	47, 508
Text	*string*	19
Comment	*comment string*	19
Definition	*expression*	47, 215
Property	NoSave, Sos1, Sos2, IndicatorConstraint Level, Bound, Basic, ShadowPrice, RightHandSideRange, ShadowPriceRange, IsDiversificationFilter, IsRangeFilter, IncludeInLazyConstraintPool, IncludeInCutPool, Chance	34, 47 217
SosWeight	*sos-weights*	
ActivatingCondition	*expression*	
Probability	*expression*	229, 337
Aproximation	*element-expression*	229, 338

Table 14.2: Constraint attributes

The Definition attribute

With the Definition attribute of a constraint you specify a numerical relationship between variables in your model. Without a definition a constraint is indeterminate. Constraint definitions consist of two or three expressions separated by one of the relational operators “=”, “>=” or “<=”.

Example

The following constraints express the simultaneous requirements that the sum of all transports from a city i must not exceed Supply(i), and that for each city j the Demand(j) must be met.

```
Constraint SupplyConstraint {
    IndexDomain  : i;
    Unit         : kton;
    Definition   : sum( j, Transport(i,j) ) <= Supply(i);
}
Constraint DemandConstraint {
    IndexDomain  : j;
    Unit         : kton;
    Definition   : sum( i, Transport(i,j) ) >= Demand(j);
}
```

Allowed relationships

If a and b are expressions consisting of only parameters and $f(x,\dots)$ and $g(x,\dots)$ are expressions containing parameters and variables, the following two kinds of relationships are allowed.

$$a \le f(x,\dots) \le b \qquad \text{or} \qquad f(x,\dots) \gtreqless g(x,\dots)$$

where $\gtreqless$ denotes any of the relational operators "=", ">=" or "<=". Either a or b can be omitted if there is no lower or upper bound on the expression $f(x,\dots)$, respectively. When both a and b are present, the constraint is referred to as a *ranged* constraint. The expressions may have linear and nonlinear terms, and may utilize the full range of intrinsic functions of Aimms except for the random number functions.

Conditional expressions in constraints

You must take extreme care to ensure continuity when the constraints in your model contain logical conditions that include references to variables. Such constraints are viewed by Aimms as nonlinear constraints, and thus can only be passed to a solver that can handle nonlinearities. It is possible that the outcome of a logical condition, and thus the form of the constraint, changes each time the underlying solver asks Aimms for function values and gradients. For example, if x(i) is a decision variable, and a constraint contains the expression

```
sum[ i, if ( x(i) > 0 ) then  x(i)^2 endif ]
```

it may or may not contain the term x(i)^2, depending on the current value of x(i). In this example, both the expression and its gradient are continuous functions at x(i) = 0.

14.2.1 Constraint properties

The Property attribute

With the Property attribute you can specify further characteristics of the constraint at hand. The possible properties of a constraint are NoSave, Sos1, Sos2, Level, Bound, Basic, ShadowPrice, RightHandSideRange, and ShadowPriceRange.

The NoSave property

When you specify the NoSave property you indicate that you do not want Aimms to store data associated with the constraint in a case, regardless of the specified case identifier selection.

14.2.2 SOS properties

The SOS properties

The constraint types Sos1 and Sos2 are used in mixed integer programming, and mutually exclusive. In the context of mathematical programming SOS is an acronym for Special Ordered Sets. A SOS set is associated with every (individual) constraint of type Sos1 or Sos2.

Additional SOS attribute

When you specify that a constraint is of type Sos1 or Sos2, an additional SOS-specific attributes becomes available, namely the SosWeight attributes. With this attributes, you can provide further information to the solver about the contents and ordering of the SOS set to be associated with the constraint.

Sos1 constraints

A type `Sos1` constraint specifies to the solver that at most one of the variables within the SOS set associated with the constraint is allowed to be nonzero, while all other variables in the SOS set must be zero. Inside a `Sos1` constraint all variables in the SOS set must have a lower bound of zero and an upper bound greater than zero.

Sos2 constraints

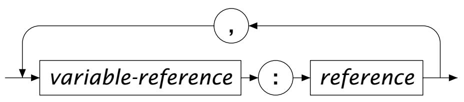
A type `Sos2` constraint specifies to the solver that at most two consecutive variables within the SOS set associated with the constraint are allowed to be nonzero, while all other variables within the SOS set must be zero. All individual variables within the SOS set must have a lower bound of zero and an upper bound greater than zero. The order of the individual variables within the SOS set is determined by their weights (as specified in the `SosWeight` attribute), where the ordering is from low to high weight.

The `SosWeight` attribute

With the `SosWeight` attribute you must specify the contents of the SOS set to be associated with a `Sos1` or `Sos2` constraint, as well the ordering of its elements. Section 7.5 of the AIMMS book on Optimization Modeling describes how these weights are used during the branch-and- bound process. The syntax of the `SosWeight` attribute is as follows.

Syntax

sos-weights :



Within the `SosWeight` attribute you can (but need not) specify a weight for every variable occurring in the constraint. Each weight must be an expression using all the indices in the index domain of the variable plus some or all of the indices in the index domain of the constraint. All weights specified for a particular constraint must be unique, i.e. you cannot specify the same weight for two (individual) variables. The SOS set to be associated with the constraint will be constructed from all variables—within the domain of both the constraint and variable—for which a nonzero weight has been specified in the `SosWeight` attribute, i.e. if the value of the specified weight is `0.0` for a particular tuple, the corresponding individual variable will not be included in the SOS set. The ordering of variables within the SOS set is from low to high weight.

Consistency

If you do not specify SOS weights, AIMMS will make sure that ordering of variables in each SOS set is consistent over all SOS sets. If you specify SOS weights yourself, you have to make sure that the variable orderings of all SOS sets of type `Sos2` are consistent, or your model might become infeasible if feasibility requires that two adjacent variables in one SOS set become nonzero, which are ordered inconsistently in another SOS set. Therefore, AIMMS requires that you specify the SosWeight attributes for *all* SOS constraints in your model, whenever you specify it for *one* SOS constraint.

Example

The following is specification of Sos2 constraint to determine the variable y piece-wise linearly from a variable x(i).

```
Constraint DetermineY {
    Property     : Sos2;
    Definition   : y = sum[ i, x(i)*c(i) ];
    SosWeight    : x(i) : XWeight(i);
}
```

14.2.3 Solution pool filtering

Solution pool

During the solution process of a MIP problem, the solvers CPLEX and GUROBI are capable of storing multiple feasible integer solutions in a solution pool, for instance, to capture solutions with attractive properties that are hard to express in a linear fashion.

Filtering

While populating the solution pool, CPLEX offers advanced filtering capabilities, allowing you to control which solutions end up in the solution pool. CPLEX provides two predefined ways to filter solutions:

- if you want to filter solutions based on their difference as compared to a reference solution, use a *diversity* filter, or
- if you want to filter solutions based on their validity in an additional linear constraint, use a *range* filter.

To enable filters the CPLEX option Do_Populate need to be on.

Diversity filters

A diversity filter allows you to generate solutions that are similar to (or different from) a set of reference values that you specify for a set of binary variables. In particular, you can use a diversity filter to generate more solutions that are similar to an existing solution or to an existing partial solution. Several diversity filters can be used simultaneously, for example, to generate solutions that share the characteristics of several different solutions.

The IsDiversificationFilter property

In AIMMS, a constraint is used as a diversity filter if the constraint property IsDiversificationFilter has been set. In a diversification filter, the Abs function is used to measure the distance from a given binary variable, and all variables should only occur as the argument of an Abs function.

Example

This following diversification filter forces the solutions to have a distance of at least 1 from variable x.

```
Constraint filter1 {
    Property     :  IsDiversificationFilter;
    Definition   :  Abs(x - 1) >= 1;
}
```

Range filters

A range filter allows you to generate solutions that obey a new constraint, specified as a linear expression within a range. Range filters can be used to express diversity constraints that are more complex than the standard form implemented by diversity filters. In particular, range filters also apply to general integer variables, semi-integer variables, continuous variables, and semi-continuous variables, not just to binary variables.

The `IsRangeFilter` property

In AIMMS, a constraint is used as a range filter if the constraint property `IsRangeFilter` has been set for the constraint.

Example

The following range filter specifies that any solution to be added to the solution pool should satisfy the following constraint.

```
Contraint filter2 {
   Property     :  IsRangeFilter;
   Definition   :  x + y + z >= 2;
}
```

14.2.4 Indicator constraints, lazy constraints and cut pools

Indicator constraints

An indicator constraint is a new way of controlling whether or not a constraint takes effect, based on the value of a binary variable. Traditionally, such relationships are expressed by so-called big-M formulations. Big-M formulations, however, can introduce unwanted side-effects and numerical instabilities into a mathematical program. Using indicator constraints, such relationships between a constraint and a variable can be directly expressed in the constraint declaration. Indicator constraints are supported by the solvers CPLEX, GUROBI and ODH-CPLEX.

The `IndicatorConstraint` property

You can specify that a constraint is an indicator constraint by settings it `IndicatorConstraint` property. For indicator constraints, a new attribute called `ActivatingCondition` will become available in the constraint declaration.

The ActivatingCondition attribute

Through the `ActivatingCondition` attribute you can specify under which condition the constraint definition should become active during the solution process. Its value should be an expression of the form

binary-variable = *expression*

where the *expression* must take one of the values 0 or 1.

Example

Consider the following big-*M* constraint

```
Constraint BigMConstraint {
    Definition : x1 + x2 <= M*y;
}
```

where `y` is a binary variable. Using the `IndicatorConstraint` property, the constraint can be reformulated as an indicator constraint as follows

```
Constraint NonBigMConstraint {
    Property            : IndicatorConstraint;
    ActivatingCondition  : y = 0;
    Definition           : x1 + x2 = 0;
}
```

The constraint only becomes effective, whenever the binary variable `y` takes the value 0. To solve the model with the indicator constraint, you need the Cplex, Gurobi or Odh-Cplex solver.

Lazy constraints

Sometimes, for a MIP formulation, a user can already identify a group of constraints that are unlikely to be violated (lazy constraints). Simply including these constraints in the original formulation could make the LP subproblem of a MIP optimization very large or too expensive to solve. Cplex, Gurobi and Odh-Cplex can handle problems with lazy constraints more efficiently, and therefore Aimms allows you to identify lazy constraints in your model.

The IncludeInLazyConstraintPool property

You can specify that a constraint should be added to the pool of lazy constraints considered by Cplex, Gurobi or Odh-Cplex by setting the property `IncludeInLazyConstraintPool`. You need the Cplex, Gurobi or Odh-Cplex solver to use this constraint property. When solving your MIP model, Cplex, Gurobi and Odh-Cplex will only consider these constraints when they are violated.

User cut pools

As discussed in Section 15.2, Aimms allows you to add cuts to your mathematical program on the fly during the solution process by using the `CallbackAddCut` callback. However, when the set of cuts you want to generate is fixed and known upfront, using the `CallbackAddCut` may add significant overhead to the solution process of your model while you don't need its flexibility. For those situations, Cplex allows you to specify a fixed pool of user cuts during the generation of your mathematical program.

The `IncludeInCutPool` property

By setting the constraint property `IncludeInCutPool` you can indicate that this constraint should be included in the pool of user cuts associated with your mathematical program. You need the Cplex solver to use this constraint property. When solving your MIP model, Cplex will consider the user cuts added in this manner when appropriate.

14.2.5 Constraint levels, bounds and marginals

Constraint levels and bounds

A constraint in Aimms can conceptually be divided such that one side consists of all variable terms, whereas the other side consists of all remaining constant terms. The *level* value of a constraint is the accumulated value of the variable terms, while the constant terms make up the *bound* of the constraint.

The `Level`, `Bound`, `Basic` and `ShadowPrice` properties

With the `Level`, `Bound`, `Basic` and `ShadowPrice` properties you indicate whether you want to store (and have access to) particular parametric data associated with the constraint.

- When you specify the `Level` property Aimms will retain the level values of the constraint as provided by the solver. You can access the level values of a constraint by using the constraint name as if it were a parameter.
- By specifying the `Bound` property, Aimms will store the upper and lower bound of the constraint as employed by the solver. You get access to the bounds by using the `.Lower` and `.Upper` suffices with the constraint identifier.
- If the `Basic` property has been specified, Aimms stores basic information is available through the `.Basic` suffix as an element in of the predefined Aimms set `AllBasicValues`. A constraint is said to be basic (nonbasic or superbasic) if its associated slack variable is basic (nonbasic or superbasic).
- With the `ShadowPrice` property you indicate that you want to store the shadow prices as computed by the solver. You can access these shadow prices by means of the `.ShadowPrice` attribute.

Interpretation of shadow prices

The shadow price (or dual value) of a constraint is the marginal change in the objective value with respect to a change in the right-hand side (i.e. the constant part) of the constraint. This value is determined by the solver after a `SOLVE` statement has been executed. The precise mathematical interpretation of the shadow price is discussed in detail in many text books on mathematical programming. Note: if a basic or superbasic constraint has a shadow price of zero then it will be displayed as 0.0, but if a nonbasic constraint has a shadow price of zero then it will be displayed as `ZERO`.

Unit of shadow price

When the variables and constraints in your model have an associated unit (see Chapter 32), special care is required in interpreting the values returned through the `.ShadowPrice` suffix. To obtain the shadow price in terms of the units specified in the model, the values of the `.ShadowPrice` suffix must be scaled as explained in Section 32.5.1.

The property `RightHandSideRange`

By specifying the `RightHandSideRange` property you request AIMMS to conduct a first type of sensitivity analysis on this constraint during a `SOLVE` statement of a linear program. The result of the sensitivity analysis are three parameters defined over the domain of the constraint. The values assigned to the parameters will be the smallest, nominal, and largest values for the *right- or left-hand side* of the constraint so that the basis remains constant. There are three cases:

- if the constraint is single sided (i.e. $f(x) \leq a$) then the smallest, nominal, and largest value for the constraint side are reported (both when constraint is binding and not binding)
- if the constraint is of range type (i.e. $a \leq f(x) \leq b$) and it is binding at one side, then the smallest, nominal, and largest value for the binding side of the constraint are reported
- if the constraint is of range type (i.e. $a \leq f(x) \leq b$) and it is not binding at neither side, then the lowest upper bound and the highest lower bound are reported.

The values are accessible through the suffices `.SmallestRightHandSide`, `.NominalRightHandSide`, and `.LargestRightHandSide`.

The property `ShadowPriceRange`

With the `ShadowPriceRange` property you request AIMMS to conduct a second type of sensitivity analysis on this constraint during a `SOLVE` statement of a linear program. The result of the sensitivity analysis are two parameters defined over the domain of the variable. The values assigned to the parameters will be the smallest and largest values that the *shadow price* of the constraint can take while holding the objective value constant. The smallest and largest values of the constraint marginals are accessible through the suffices `.SmallestShadowPrice` and `.LargestShadowPrice`.

Linear programs only

As with the advanced sensitivity properties of variables (see Section 14.1.2), AIMMS also supports the advanced sensitivity analysis conducted through the properties `RightHandSideRange` and `ShadowPriceRange` for linear mathematical programs only. Again, if you want to apply these types of analysis to the final solution of a mixed-integer program, you should fix all integer variables to their final solution (using the `.NonVar` suffix) and re-solve the resulting mathematical program as a linear program.

Storage and computational costs

Setting any of the properties ShadowPrice, ShadowPriceRange or RightHandSide-Range may result in an increase of the memory usage. In addition, the computations required to compute the ShadowPriceRange may considerably increase the total solution time of your mathematical program.

14.2.6 Constraint suffices for global optimization

Suffices for global optimization

AIMMS provides a number of constraint suffices especially for the global optimization solver BARON. They are:

- the .Convex suffix, and
- the .RelaxationOnly suffix.

By providing additional knowledge, that cannot be determined automatically by BARON itself, about the constraints in your model through these suffices, the BARON solver may be able to optimize your global optimization model in a more efficient manner. For more detailed information about the specific capabilities of the BARON solver, you are referred to the BARON website http://www.theoptimizationfirm.com/.

The .Convex suffix

The algorithm of the BARON solver exploits convexity—either identified automatically by BARON itself or explicitly supplied in the model formulation—in order to generate polyhedral cutting planes and relaxations for multivariate non-convex problems. Through the .Convex suffix you can explicitly indicate that a particular constraint is convex if BARON is unable to determine its convexity automatically.

The .RelaxationOnly suffix

Using the .RelaxationOnly suffix, you can considerably enhance the convexification capabilities of BARON. Some nonlinear problem reformulations can often tighten the relaxation process of BARON's branch-and-bound algorithm while making local search considerably more difficult. By assigning a nonzero value to the .RelaxationOnly suffix, you indicate to BARON that the constraint at hand should only be included as a relaxation to the branch-and-bound algorithm, while it should be excluded from the local search.

14.2.7 Chance constraints

Chance constraints

The AIMMS modeling language offers facilities for robust optimization models, including support for *chance constraints* (see also Section 20.3). By setting the Chance property of a constraint, the constraint will become a chance constraint when solving a mathematical program using robust optimization, using the distributions specified for the random parameters contained in its definition. When setting the Chance property, two new attributes will become available, the Probability attribute and the Approximation attribute.

Only for robust optimization

Note that setting the Chance property does not influence the availability and use of the constraint outside the context of robust optimization. In that case, AIMMS will just use the original, deterministic, constraint definition, completely disregarding the uncertainty of the parameters used in the constraint.

The Probability attribute

Through the Probability attribute, you can specify the probability with which you want the constraint to be satisfied for any feasible solution to the robust counterpart of a robust optimization model. Its value must be a numerical expression in the range $[0, 1]$.

The Approximation attribute

When constructing the robust counterpart, AIMMS can use several types of approximations to approximate the chance constraint at hand. You can use the Approximation attribute to specify the type of approximation you want to be applied. The chosen type of approximation may lead to a robust counterpart which is easier or harder to solve (see also Section 20.3). The value of the attribute must be an element expression into the predefined set AllChance-ApproximationTypes.