AIMMS Language Reference - Constraint Programming

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.

Chapter 22

Constraint Programming

Introduction

Constraint Programming is a relatively new paradigm used to model and solve combinatorial optimization problems. It is most effective on highly combinatorial problem domains such as timetabling, sequencing, and resource-constrained scheduling. Successful industrial applications utilizing constraint programming technology include the gate allocation system at Hong Kong airport, the yard planning system at the port of Singapore, and the train timetable generation of Dutch Railways.

Constraint Programming in AIMMS

This chapter discusses the special identifier types and language constructs that AIMMS offers for formulating and solving constraint programming problems. We will see that constraint programming offers a much wider range of modeling constructs than, for example, integer linear programming or nonlinear programming. Different variable types can be used, while restrictions can be formed using arbitrary algebraic and logical expressions or by the use of special constraint types, such as alldifferent. In addition, AIMMS offers a specific syntax to express scheduling problems in an intuitive way, taking advantage of the algorithmic power that underlies constraint-based scheduling.

This chapter

In this chapter, the basic constraint programming concepts are first presented, including different variable types and restrictions in Section 22.1. Section 22.2 discusses the AIMMS syntax for modeling constraint-based scheduling problems. The final section of this chapter discusses issues related to modeling and solving constraint programs in AIMMS.

Literature

An in-depth discussion on constraint programming is given in [Ro06] and more details on constraint-based scheduling can be found in [Ba01].

Online resources

First, the Association for Constraint Programming organizes an annual summer school, the material for which is posted online. This material can be accessed at `http://4c.ucc.ie/a4cp/`. The CPAIOR conference series organizes tutorials alongside each event, the materials of which are posted online. The CPAIOR 2009 tutorial provides an introduction to constraint programming and hybrid methods, and available online at `http://www.tepper.cmu.edu/cpaior09`.

22.1 Constraint Programming essentials

Variables

In constraint programming, models are built using variables and constraints, and as such is similar to integer programming. One fundamental difference is that, in integer programming, the range of a variable is specified and maintained as an interval, while in constraint programming, the variable range is maintained explicitly as a set of elements. Note that in the constraint programming literature, the range of a variable is commonly referred to as its *domain*.

Constraints

As a consequence of this explicit range representation, constraint programming can offer a wide variety of constraint types. Most constraint programming solvers allow constraints to be defined by arbitrary expressions that combine algebraic or logical operators. Moreover, meta-constraints can be formulated by interpreting the logical value of an expression as a boolean value in a logical relation, or as a binary value in an algebraic relation. For example, to express that every two distinct tasks i and j, from a set of tasks T with respective starting times s_i, s_j and durations d_i, d_j, should not overlap in time, we can use logical disjunctions:

$$(s_i + d_i \leq s_j) \vee (s_j + d_j \leq s_i), \;\; \forall i, j \in T, i \neq j. \tag{22.1}$$

As another example, we can set a restriction such that no more than half the variables from $x_1, \ldots, x_n$ are assigned to a specific value v as $\sum_{i=1}^{n}(x_i = v) \leq 0.5n$.

Global constraints

In addition, constraint programming offers special symbolic constraints that are called *global constraints*. These constraints can be defined over an arbitrary set of variables, and encapsulate a combinatorial structure that is exploited during the solving process. The constraint `cp::AllDifferent`$(x_1 \ldots, x_n)$ is an example of such a global constraint. This global constraint requires the variables $x_1 \ldots, x_n$ to take distinct values.

Solving

The solving process underpinning constraint programming combines systematic search with inference techniques. The systematic search implicitly enumerates all possible variable-value combinations, thus defining a search tree in which the root represents the original problem to be solved. At each node in the search tree, an inference is made by means of *domain filtering* and *constraint propagation*. Each constraint in the model has an associated domain filtering algorithm that removes provably inconsistent values from the variable domains. Here, a domain value is inconsistent if it does not belong to any solution of the constraint. The updated domains are then communicated to the other constraints, whose domain filtering algorithms in turn become active; this is the constraint propagation process. In practice, the most effective filtering algorithms are those associated with global constraints. Therefore, most practical applications that are to be solved with constraint programming are formulated using global constraints.

Application domains

Constraint programming can be particularly effective with highly combinatorial problem domains, such as timetabling or resource-constrained scheduling. For such problems an integer programming model may be non-intuitive to express. Moreover, the associated continuous relaxation may be quite weak, which makes it much harder to find a provably optimal solution. For example, again consider two tasks A and B that must not overlap in time. In integer programming one can introduce two binary variables, y_{AB} and y_{BA}, representing that task A must be processed either before or after, task B. The non-overlapping constraint can then be expressed as $y_{AB} + y_{BA} = 1$, for which a continuous linear relaxation may assign a solution $y_{AB} = y_{BA} = 0.5$, which is not very informative. In contrast, the non-overlapping requirement can be handled very effectively using a specific ‘sequential resource' scheduling constraint in constraint programming that effectively groups together all pairwise logical disjunctions in (22.1) above. Such a constraint is also referred to as a disjunctive or unary constraint in the constraint programming literature.

Designing models

The expressiveness of constraint programming offers a powerful modeling environment, albeit one that comes with a caveat. Namely, that different syntactically equivalent formulations may yield quite different solving times. For example, an alternative to the constraint `cp::AllDifferent`$(x_1, x_2, \ldots, x_n)$ is its decomposition into pairwise not-equal constraints $x_i \neq x_j$ for $1 \leq i < j \leq n$. The domain filtering algorithm for `cp::AllDifferent` provably removes more inconsistent domain values than the individual not-equal constraints, which results in much smaller search trees and faster solution times. Therefore, when designing a constraint programming model, one should be aware of the effect that different formulations can have on the solving time. In most situations, it is advisable to apply global constraints to exploit their algorithmic power.

22.1.1 Variables in constraint programming

Variables of a constraint program

A constraint programming problem is made up of discrete variables and constraints over these discrete variables. A discrete variable is a variable that takes on a discrete value. Aimms supports two base types of discrete variables for constraint programming. The first type of variable is the integer variable; an ordinary variable with a range formulated such as $\{$`a..b`$\}$ where `a` and `b` are numbers or references to parameters (see Chapter 14). Such variables can also be used in `MIP` problems. The second type of variable is the element variable, which will be further detailed in this section. This type of variable can only be used in a constraint programming problem. These two types of variable can be combined in a third type, called an integer element variable, which supports the operations that are defined for both the integer variable and the element variable.

22.1.1.1 ElementVariable declaration and attributes

An element variable is a variable that takes an element as its value. It can have the attributes specified in Table 22.1. The attributes `IndexDomain`, `Priority`, `NonvarStatus`, `Text`, `Comment` are the same as those for the variables introduced in Chapter 14.

Attribute	**Value-type**	**See also page**
`IndexDomain`	*index-domain*	212
`Range`	*named set*	
`Default`	*constant-expression*	46, 214
`Priority`	*expression*	215
`NonvarStatus`	*expression*	216
`Property`	`NoSave`	34
`Text`	*string*	19
`Comment`	*comment string*	19
`Definition`	*expression*	215

Table 22.1: `ElementVariable` attributes

The `Range` attribute

The range of an element variable is a one-dimensional set, similar to the range of an element parameter. This attribute must be a set identifier; and this permits the compiler to verify the semantics when element variables are used in expressions. This attribute is mandatory.

The `Default` attribute

The attribute `default` of an element variable is a quoted element. This attribute is not mandatory.

The `Definition` attribute

The `Definition` attribute of an element variable is similar to the definition attribute of a variable, see also page 215, except that its value is an element and the resulting element must lie inside the range of the element variable. This attribute is not mandatory.

The `Property` attribute

The following properties are available to element variables:

- **Nosave** When set, this property indicates that the element variable is not to be saved in cases.
- **EmptyElementAllowed** When set, this property indicates that in a feasible solution, the value of this variable can, but need not, be the empty element `''`. When the range of the element variable is a subset of the set

Integers, this property is not available. In the following example, for the element variable eV, not selecting an element from S, is a valid choice, but this choice forces the integer variable X to 0.

```
ElementVariable eV {
    Range      :  S;
    Property   :  EmptyElementAllowed;
}
Constraint Force_X_to_zero_when_no_choice_for_eV {
   Definition   :  if eV = '' then X = 0 endif;
}
```

This attribute is not mandatory.

Element translation

Constraint programming solvers only use integer variables, and AIMMS translates an element variable, say eV with range the set S containing n elements into a integer variable, say v, with range $\{0..n-1\}$. By design, this translation leaves no room for the empty element '', and subsequently, in a feasible solution, the empty element is no part of it. In order to permit the explicit consideration of the empty element as part of a solution, the property EmptyElementAllowed can be set for eV. In that case the range of v is $\{0..n\}$ whereby the value 0 corresponds to the empty element.

22.1.1.2 Selecting a variable type

Choosing variable type

When there are multiple types of objects, such as integer variables and element variables in AIMMS, the following two questions naturally arise:

1. How to choose between the various types?
2. Can these types be combined?

The answers to these questions are as follows:

1. You may want to base the choice of types of variables on the operations that can be performed meaningfully on these types. Which operation is appropriate for which variable type is described below.
2. An identifier can have both the 'integer variable' and 'element variable' types and is then called an 'integer element variable'. This is created as an element variable with a named subset of the predeclared set Integers as its range.

Operations on variables

The operations on variables that are interesting in constraint programming are:

- **Numeric** operations, such as multiplication, addition, and taking the absolute value. These operations are applicable to integer variables and to integer element variables.
- **Index** operations; selecting an element of an indexed parameter or variable. An element variable eV can be an argument of a parameter P or

a variable X in expressions such as P(eV) or X(eV). These operations are applicable to all element variables. In the Constraint Programming literature, such operations are often implemented using so-called element constraints.

- **Compare, subtract, min and max** operations. These operations are applicable to all discrete variables, including element variables. For element variables, AIMMS uses the ordering of sets, see Section 3.2.

All of the above operations are available with integer element variables.

Contiguous range

In order to limit an element variable to a contiguous subset of its named range, element valued suffixes `.lower` and `.upper` can be used. In the example below, the assignment to `eV.Lower` restricts the variable `eV` to the contiguous set {`c..e`}.

```
Set someLetters {
    Definition   : data { a, b, c, d, e };
}
ElementVariable eV {
    Range        : someLetters;
}
Procedure Restrict_eV {
    Body         : eV.lower := 'c';
}
```

The specification of non-contiguous ranges, informally known as ranges with holes, is detailed in the next subsection.

22.1.2 Constraints in constraint programming

Introduction

The constraints in constraint programming allow a rich variety of restrictions to be placed on the variables in a constraint program, ranging from direct domain restrictions on the variables to global constraints that come with powerful propagation algorithms.

Domain restrictions

A domain restriction restricts the domain of a single variable, or of multiple variables, and is specified using the `IN` operator. For example, we can restrict the domain of an element variable `eV` as follows:

```
Constraint DomRestr1 {
    Definition   :  eV in setA;
}
```

When we apply the `IN` operator to multiple variables, we can define a constraint by explicitly listing all tuples that are allowed. For example:

```
Constraint DomRestr2 {
    Definition   :  (eV1, eV2, eV3) in ThreeDimRelation;
    Comment      : "ThreeDimRelation contains all allowed tuples";
}
```

```
Constraint DomRestr3 {
    Definition   :  not( (eV1, eV2, eV3) in ComplementRelation );
    Comment      : "ComplementRelation contains all forbidden tuples";
}
```

In constraint DomRestr2 above, the three element variables are restricted to elements from the set of allowed tuples defined by ThreeDimRelation. Alternatively, we can define such a restriction using the complement, i.e., a list of forbidden tuples, as with constraint DomRestr3. In constraint programming, these constraints are also known as *Table* constraints; the data for these constraints resemble tables in a relational database.

Algebraic restrictions

The following operations are permitted on discrete variables, resulting in expressions that can be used in constraint programming constraints:

1. The binary min(a,b), max(a,b) and the iterative min(i,x(i)), max(i,x(i)) can both be used,
2. multiplication *, addition +, subtraction -, absolute value abs and square sqr,
3. integer division div(a,b), integer modulo mod(a,b),
4. floating point division /, and
5. indexing: an element variable is used as an argument of another parameter or variable, P(eV), V(eV),

Note that the operation must be meaningful for the variable type, see page 370. These expressions can be compared, using the operators <=, <, =, <>, >, and >= to create algebraic restrictions. Simple examples of algebraic constraints, taken from Einstein's Logic Puzzle, are presented below.

```
Constraint Clue15 {
    Definition   :  abs( Smoke('Blends') - Drink('Water') ) = 1;
    Comment      :  "The man who smokes Blends has a neighbor who drinks water.";
}
Constraint TheQuestion {
    Definition   :  National(eV)=Pet('Fish');
    Comment      :  "Who owns the pet fish? Result stored in element variable eV";
}
```

Combining restrictions

The constraints above can be combined to create other constraints called meta-constraints. Meta-constraints can be formed by using the scalar operators AND, OR, XOR, NOT and IF-THEN-ELSE-ENDIF. For example:

```
Constraint OneTaskComesBeforeTheOther {
    Definition   : {
        ( StartA + DurA <= StartB ) or
        ( StartB + DurB <= StartA )
    }
}
```

In addition, restrictions can be combined into meta-constraints using the iterative operators FORALL and EXISTS. Moreover, restrictions can be counted using the iterative operator SUM and the result compared with another value. Finally, meta-constraints are restrictions themselves, they can be combined into even more complex meta-constraints. The following example is a variable definition, in which the collection of constraints (Finish(i) > Deadline(i)) is used to form a meta-constraint.

```
Variable TotalTardinessCost {
    Definition  :  Sum( i, TardinessCost(i) | ( Finish(i) > Deadline(i) ) );
}
```

In the following example, the binary variable y gets the value 1 if each X(i) is greater than P(i).

```
Constraint Ydef {
    Definition   :  y = FORALL( i, X(i) > P(i) );
}
```

From the Steel Mill example, we can model that we do not want more than two colors for each slab by the following nested usage of meta-constraints:

```
Constraint EnhancedColorCst {
    IndexDomain  :  (sl);
    Definition   :  sum( c, EXISTS (o in ColorOrders(c), SlabOfOrder(o)=sl)) <= 2;
}
```

Global constraints

AIMMS supports the global constraints presented in Table 22.2. These global constraints come with powerful filtering techniques that may significantly reduce the search tree and thus the time needed to solve a problem.
The example below illustrates the use of the global constraint cp::AllDifferent as used in the Latin square completion problem. A Latin square of order n is an $n \times n$ matrix where the values are in the range $\{1..n\}$ and distinct over each row and column.

```
Constraint RowsAllDifferent {
    IndexDomain  :  r;
    Definition   :  cp::AllDifferent( c, Entry(r, c) );
}
Constraint ColsAllDifferent {
    IndexDomain  :  c;
    Definition   :  cp::AllDifferent( r, Entry(r, c) );
}
```

Additional examples of global constraints are present in the AIMMS Function Reference. Unless stated otherwise in the function reference, global constraints can also be used outside of constraints definitions, for example in assignments or parameter definitions.

Global constraint	**Meaning**
`cp::AllDifferent`(i,x_i)	The x_i must have distinct values. $\forall i,j \mid i \neq j : x_i \neq x_j$
`cp::Count`(i,x_i,c,$\otimes$,y)	The number of x_i related to c is y. $\sum_i (x_i = c) \otimes y$ where $\otimes \in \{\leq, \geq, =, >, <, \neq\}$
`cp::Cardinality`(i,x_i, j,c_j,y_j)	The number of x_i equal to c_j is y_j. $\forall j : \sum_i (x_i = c_j) = y_j$
`cp::Sequence`(i,x_i, S,q,l,u)	The number of $x_i \in S$ for each subsequence of length q is between l and u. $\forall i = 1..n-q+1 :$ $l \leq \sum_{j=i}^{i+q-1} (x_j \in S) \leq u$
`cp::Channel`(i,x_i, j,y_j)	Channel variable $x_i \to J$ to $y_j \to I$ $\forall i,j : x_i = j \Leftrightarrow y_j = i$
`cp::Lexicographic`(i,x_i,y_i)	x is lexicographically before y $\exists i : \forall j < i : x_j = y_j \wedge x_i < y_i$
`cp::BinPacking`(i,l_i, j,a_j,s_j)	Assign object j of known size s_j to bin $a_j \to I$. Size of bin $i \in I$ is l_i. $\forall i : \sum_{j \mid a_j = i} s_j \leq l_i$

Table 22.2: Global constraints

Global constraint vector arguments

These global constraints have vectors as arguments. The size of a vector is defined by a preceding index binding argument. Further information on index binding can be found in the Chapter on Index Binding 9. Such a vector can be a vector of elements, for example the fourth argument of `cp::Cardinality`. In a vector of elements, the empty element `''` is not allowed; comparison of an element variable against the empty element is not supported.

Basic scheduling constraints

Aimms offers support for both basic scheduling and advanced scheduling. Advanced scheduling will be detailed in the next section but, for basic scheduling, Aimms offers the following two global constraints:

1. The global constraint `cp::SequentialSchedule`(j, s_j, d_j, e_j) ensures that two distinct jobs do not overlap where job j has start time s_j, duration d_j and end time e_j. This constraint is equivalent to:
   - $\forall i,j, i \neq j : (s_i + d_i \leq s_j) \vee (s_j + d_j \leq s_i)$.
   - $\forall j : s_j + d_j = e_j$

   This and similar constraints are also known as `unary` or `disjunctive` constraints within the Constraint Programming literature.
2. The global constraint `cp::ParallelSchedule`(l, u, j, s_j, d_j, e_j, h_j) allows a single resource to handle multiple jobs, within limits l and u, at the same time. Here job j has start time s_j, duration d_j, end time e_j and

resource consumption (height) h_j. This constraint is equivalent to:

- $\forall t : l \le \sum_{j | s_j \le t < e_j} h_j \le u$.
- $\forall j : s_j + d_j = e_j$

This and similar constraints are also known as `cumulative` constraints within the Constraint Programming literature.

22.2 Scheduling problems

Introduction

Resource-constrained scheduling is a key application area of constraint programming. Most constraint programming systems contain special syntactical constructs to formulate such problems, allowing the use of specialized inference algorithms. In Section 22.1.2 we have already seen two examples of global constraints for scheduling: `cp::SequentialSchedule` and `cp::ParallelSchedule`. For more complex scheduling problems that contain, for example, sequence-dependent setup times between activities, or specific precedence relations, the use of more advanced scheduling algorithms is advisable. These algorithms cannot be offered by the stand-alone global constraints `cp::SequentialSchedule` and `cp::ParallelSchedule`, but can be accessed by formulating the problem using *activities* and *resources* in AIMMS.

Activities

Activities correspond to the execution of objects in scheduling problems, e.g., processing an order, working a shift, or performing a loading operation. They can be viewed as the variables of a scheduling problem, since we must decide on their position in the schedule. Common attributes associated to an activity are its begin, end, length, and size. Further, it is often convenient to distinguish mandatory and optional activities, which allows to consider the presence of an activity. In AIMMS, the properties begin, end, length, size, and presence of an activity can be used as variables in other parts of the model. It is also possible to build models using nested activities, where meta-activities group together a number of sub-activities, for example in the context of project planning.

Resources

Resources correspond to the assets that are available to execute the activities, e.g., the capacity of a machine, the volume of a truck, or the number of available employees. Resources can be viewed as the constraints of a scheduling problem. The main attributes of a resource are its capacity, its activity level, and the set of activities that require the resource in order to be executed. That is, during the execution of the schedule, we must ensure that the resource activity level is always within its capacity. Note that while a resource depends on a set of activities, an activity can impact on one or more resources at the same time.

An activity changes the resource activity level

A resource starts with a default activity level of 0, corresponding to full available capacity, or a user-specified initial value. During the execution of the schedule, activities will influence the resource activity level. The viewpoint chosen in AIMMS is that an activity changes the activity level of a resource when it begins, and/or when it ends. This enables one to model many common situations. For example, when an activity corresponds to a loading operation, and the resource corresponds to a truck load, the activity will change the activity level of the resource with the volume of the load at its start, but there is no change in the resource activity level when finishing this activity. For sequential resources the capacity is 1, and each activity will change the resource activity level by $+1$ when it begins, and by -1 when it ends. For example, when an activity corresponds to a visit operation, and the resource corresponds to a truck, the activity level of the resource will be decreased by 1 at the beginning of the visit, and increased by 1 at the end.

Schedule domains

The timeline on which activities are scheduled, is the so-called *schedule domain*. A schedule domain is a finite set of timeslots. Each activity and resource has its own schedule domain.

The problem schedule domain

The schedule domain of the entire problem, the *problem schedule domain*, is a named superset of each of these schedule domains. Unless overridden, it is based on the schedule domains of the activities and resources.

Handling schedule domains

An activity is only considered active during a timeslot t if t is in the schedule domain of that activity, and it is in the schedule domain of each resource for which it is scheduled. Thus, for each individual activity, AIMMS passes the intersection of these schedule domains to the constraint programming solver.

Additional restrictions

Most scheduling problems contain several side constraints in addition to the resource constraints. Examples include precedence relations between activities, release dates or deadlines, and sequence-dependent setup times. Such constraints can be specified using global scheduling constraints or in the attribute forms of activities and resources. Constraint programming solvers can take an extra algorithmic advantage of such constraints when they are presented in this manner.

22.2.1 Activity

On the one hand, an activity can be seen as consisting of five variables that can be accessed by the suffixes: `.Begin`, `.End`, `.Length`, `.Size` and `.Present`. These variables represent the begin, end, length (difference between end and begin), size (number of active slots) and presence of an activity. These variables can be used inside constraints, for example `myActivity.End <= myDeadLine+1`. On the other hand, an activity is defined using its attributes as presented in Table 22.3. We will first discuss the attributes of an activity, and then these suffixes in more detail.

Attribute	**Value-type**	**See also page**
IndexDomain	*index-domain*	212
ScheduleDomain	*set range or expression*	
Property	Optional, NoSave	
Length	*expression*	
Size	*expression*	
Priority	*reference*	215
Text	*string*	19
Comment	*comment string*	19

Table 22.3: Activity attributes

The ScheduleDomain attribute

The activity is scheduled in time slots in the ScheduleDomain. This is an expression resulting in a one-dimensional set, or a set-valued range. The resulting set need not be a subset of the predeclared set Integers; it can be any one-dimensional set, for instance a Calendar, see Section 33.2. Consider the following examples of the attribute schedule domain:

```
Activity a {
    ScheduleDomain  :  yearCal;
    Comment         : {
        "a can be scheduled during any period
        in the calendar yearCal"
    }
}
Activity b {
    IndexDomain     :  i;
    ScheduleDomain  :  possiblePeriods(i);
    Comment         : {
        "b(i) can be scheduled only during the
        periods possiblePeriods(i)"
    }
}
Activity c {
    IndexDomain     :  i;
    ScheduleDomain  : {
        {ReleaseDate(i)..PastDeadline(i)}
    }
    Comment         : {
        "c(i) must start on or after ReleaseDate(i)
         c(i) must finish before PastDeadline(i)"
    }
}
```

The ScheduleDomain attribute is mandatory.

Singleton schedule domain

An activity with a singleton schedule domain and a length of 1 can be used to model an event. Such an activity is scheduled during the single element in the schedule domain. Because the schedule domain is a single element, the value of the suffixes `.Begin` and `.End` of the activity will be set to that single element and the element thereafter respectively in a feasible solution. Note that this is possible for all elements except for the last element in the problem schedule domain; a nonzero length would then require the `.End` to be after the problem schedule domain. Consider the following example:

```
Activity weekendActivities {
    IndexDomain     : {
        d | ( TimeslotCharacteristic( d, 'weekday' ) = 6 or
        TimeslotCharacteristic( d, 'weekday' ) = 7    ) and
        d <> last( dayCalendar )
    }
    ScheduleDomain  : {
        { d .. d }
    }
    Length          :  1;
    Comment         :  "d is an index in a calendar";
}
```

Scheduling the activity `weekendActivities` in a sequential resource will block other activities for that resource during the weekend.

The `Property` attribute

An activity can have the properties `Optional`, `Contiguous` and `NoSave`.

- **`Optional`** When an activity has the property `Optional`, it may or may not be scheduled. If the property `Optional` is not specified, then the activity will always be scheduled.
- **`Contiguous`** When an activity has the property `Contiguous`, all elements from `.Begin` up to but not including `.End` in the problem schedule domain must be in its own schedule domain.
- **`NoSave`** When an activity has the property `NoSave`, it will not be saved in cases.

This attribute is not mandatory.

The `Length` and `Size` attributes

When an activity is present, the `Length` attribute defines the length of the activity, and the `Size` attribute defines its size. The length of an activity is the difference between its end and its begin. The size of an activity is the number of periods, in which that activity is active from begin up to but not including its end. For example, a non-contiguous activity which `.Begin`s on Friday, `.End`s on Tuesday, and is not active during the weekend has a

- `.Length` of 4 days, and
- `.Size` of 2 days.

The numeric expressions entered at the `Length` and `Size` attributes may involve other discrete variables. These attributes are not mandatory.

For a contiguous activity we have that the .Length is equal to the .Size. Conversely, with a constraint a.Length=a.Size we have that a is contiguous, but the propagation may be less efficient.

The Priority attribute

The Priority attribute applies to all the discrete variables defined by an activity. To these variables it has the same meaning as for integer variables, see page 215. This attribute is not mandatory.

The suffixes of activities

An activity is made up of the following suffixes .Begin, .End, .Length, .Size and .Present. Each of these suffixes is a discrete variable and can be used in constraints.

The suffixes .Begin and .End

The suffixes .Begin and .End are element valued variables. When scheduled, the activity takes place from period .Begin up to but not including period .End. For a present activity a, in a feasible solution:

- a.Begin is an element in the schedule domain of the activity. The range of this element variable is the *smallest* named set encompassing the activity schedule domain.
- a.End is an element in the schedule domain of the problem, and, depending on the .Length of a, with the following additional requirement:
  - When the length of activity a is zero, a.End=a.Begin holds, and they are both in the activity schedule domain.
  - When the length of activity a is greater than 0, the element before a.End is in the activity schedule domain.

  The range of this element variable is the *root* set of the activity schedule domain.

Comparison of the .Begin and .End suffixes of two activities a and b inside a constraint definition will take place on the problem schedule domain, for instance in a constraint like a.End <= b.Begin. Outside constraint definitions these suffixes follow the rules of element comparison specified in Section 6.2.3.

The suffixes .Length and .Size

The suffixes .Length and .Size are nonnegative integer variables. The .Length of an activity is defined as .End - .Begin. The .Size of an activity is the number of timeslots in the schedule domain of the activity in the range [.Begin, .End). When the attribute Length or Size is non-empty, AIMMS will generate a defining constraint for the suffix .Length resp. .Size like the definition attribute of a variable, see Page 215. When the schedule domain of an activity is a calendar or a subset thereof, the unit of each of the .Length and .Size suffixes is the unit of the calendar.

The suffix .Present

The suffix `.Present` is a binary variable with default `0`. For optional activities this variable is 1 when the activity is scheduled and 0 when it is not. For non-optional activities this variable is initialized with the value 1.

suffixes of absent activities in constraints

The value of one of the suffixes `.Begin`, `.End`, `.Length`, and `.Size` is not defined when the corresponding activity is absent. However, in order to satisfy constraints where such a suffix is used for an absent activity, a value is chosen: the socalled *absent* value. For the suffixes `.Length`, and `.Size`, the absent value is 0. For the suffixes `.Begin` and `.End` this depends on the problem schedule domain:

- If the problem schedule domain is a subset of `Integers`, the absent value is 0.
- Otherwise, the absent value of the suffixes `.Begin` and `.End` is `''`.

To override the absent value use one of the following functions:

- `cp::ActivityBegin`,
- `cp::ActivityEnd`,
- `cp::ActivityLength`, or
- `cp::ActivitySize`.

Suffixes of optional activities in constraints

The value of the suffix `.present` is defined for an absent activity as 0. However the values of the other suffixes of an absent activity are not defined. To enable the constraining of the values of those suffixes in constraints several formulation alternatives are available. As an example of these alternatives, consider an activity `act` whereby we want to enforce its length to be 7 if it is present.

1. Enforce the length constraint conditionally on the presence of activity `act`:

```
if  act.present then
    act.length = 7
endif
```

2. The `cp::ActivityLength` function returns the length of a present activity or its second argument if it is not present:

```
cp::ActivityLength( act, 7 ) = 7
```

3. If we simply want to set the value of the `.Length` or `.Size` suffix, we can use the `Length` or `Size` attribute as follows.

```
Activity act {
    ScheduleDomain  :  ...;
    Property        :  optional;
    Length          :  7;
}
```

Each of the above formulation alternatives has its own merits.

1. The merit of this alternative is that it is general and can also be used to state for instance that the length of `act` is 7 or 11 when present:

   ```
   if  act.present then
       act.Length = 7 or act.Length = 11
   endif
   ```

2. The merit of this alternative is that it allows the solver to make stronger propagations and thus potentially reduce solution time.
3. The merit of this alternative is that is does not force the model builder to take the optionality of `act` into account when defining its length. Aimms will make sure the length definition is translated to alternative 1 or 2 as appropriate.

Solution values of absent activities

The value of the suffixes `.Begin`, `.End`, `.Length`, and `.Size` of an absent activity in a feasible schedule are meaningless and should not be used in further computations.

Absent versus 0-length activities

Even though no work is done for both absent and 0-length activities, there is a difference in their usage. Let us consider the following two examples:

- Selection of an activity from alternatives; Consider a collection of activities from which we need to select one. This is easily and efficiently achieved by setting the property `Optional` to the activity. The ones not selected become absent in a solution.
- Consider two collections of activities, whereby the n activities in the first collection all need to be completed before the m activities in the second collection can start. We can model this directly by $n \times m$ precedence constrains. Another way to model this is by introducing an extra activity, say `Milestone`, of length zero. With this `Milestone` we only need $n + m$ precedence constraints.

To facilitate above and other examples of scheduling, the suffixes `.Present` and `.Length` are supported independently.

Relation between suffixes of activities

Please note, for an activity `act`, the following relation is implicitly defined:

```
if act.Present then
   act.Begin + act.Length = act.End
endif
if act.Present then
   act.Size <= act.Length
endif
```

22.2.2 Resource

A resource schedules activities by acting as a constraint on the activities it schedules. A feasible resource requires the above implicit constraints on the suffixes of the activities it schedules and the constraints implied by its attributes as discussed below.

Attribute	Value-type	See also page
`IndexDomain`	*index-domain*	212
`Usage`	`Parallel` or `Sequential`	
`ScheduleDomain`	*set range or expression*	377
`Activities`	*collection of activities*	
`Property`	`NoSave`	
`GroupSet`	*a reference to a set*	
`GroupDefinition`	*activity : expression*	
`GroupTransition`	*index domain : expression*	
`Transition`	*set of reference pair : expression*	
`FirstActivity`	*reference*	
`LastActivity`	*reference*	
`ComesBefore`	*set of reference pairs*	
`Precedes`	*set of reference pairs*	
`Unit`	*unit-valued expression*	47, 215
`LevelRange`	*numeric range*	
`InitialLevel`	*reference*	
`LevelChange`	*per activity : expression*	
`BeginChange`	*per activity : expression*	
`EndChange`	*per activity : expression*	
`Text`	*string*	19
`Comment`	*comment string*	19

Table 22.4: `Resource` attributes

A resource is defined using the attributes presented in Table 22.4.

The `Usage` *attribute*

A resource can be used in two ways: `Parallel`, and `Sequential`, of which precisely one must be selected. The resource usage is then as follows

- **`Sequential`:** Defines the resource to be used sequentially. Such a resource is also known as a unary or disjunctive resource. A sequential resource has the additional attributes `Transition`, `FirstActivity`, `LastActivity`, `ComesBefore`, and `Precedes`, see Subsection 22.2.2.1.
- **`Parallel`:** Defines the resource to be used in parallel. Such a resource is also known as a cumulative resource. A parallel resource has the ad-

ditional attributes LevelRange, InitalLevel, LevelChange, BeginChange, and EndChange, see Subsection 22.2.2.2.

The Usage attribute is mandatory; either Sequential or Parallel must be selected.

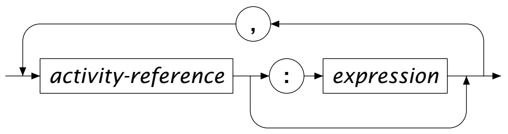
The ScheduleDomain attribute

The resource is affected by activities during the periods set in its schedule domain. This is an expression resulting in a one-dimensional set, or a set-valued range. AIMMS verifies that the schedule domain of the resource matches the schedule domain of all activities it is affected by. Here, two sets match if they have a common super set.
When the intersection of the schedule domain of a resource and the schedule domain of a non-optional activity are empty, the result is an infeasible schedule.

The Activities attribute

The Activities attribute details the activities affecting the resource. This adheres to the syntax:

activity-selection :



as illustrated in the example below:

```
Resource myMachine {
    ScheduleDomain  :  H;
    Usage           :  ...  ! sequential or parallel;
    Activities      : {
        maintenance,  ! Maintenance is scheduled between actual jobs.
        simpleJob(i), ! Every simple job can be done on this machine.
        specialJob(j) : jobpos(j) ! Only selected special jobs are allowed.
    }
}
```

In this example, the activities maintenance and simpleJob can affect the resource myMachine. However, the activity specialJob(j) can only affect the resource when jobpos(j) is non-zero. Only the detailed activities can be used in the attributes that follow. The Activities attribute is mandatory.

The Property attribute

A resource can have the properties: NoSave and TransitionOnlyNext.

- When the property NoSave is set, this indicates that the resource data will not be saved in cases.
- The property TransitionOnlyNext is relevant to the attributes Transition and GroupTransition of sequential resources only, and is discussed after the GroupTransition attribute below.

The attribute Property is not mandatory.

22.2.2.1 Sequential resources

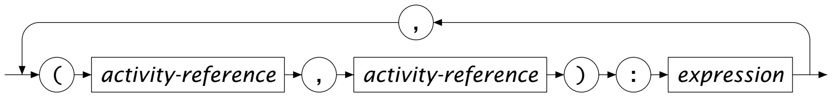
Sequential resources

Sequential resources are used to schedule activities that are not allowed to overlap. Those workers and machines that can only handle one activity at a time are typical examples. A sequential resource has only one suffix, namely `.ActivityLevel`. A sequential resource is active when it is servicing an activity, and then its `.ActivityLevel` is 1. When a sequential resource is not active, or idle, its `.ActivityLevel` is 0. The attributes particular to sequential resources are discussed below. The `.ActivityLevel` suffix cannot be used in constraint definitions.

The `Transition` attribute

The `Transition` attribute is only available to sequential resources, and then only if the `GroupSet` attribute has not been specified. This attribute contains a matrix between activities `a` and `b`, specifying the minimal time between `a` and `b` if `a` is scheduled before `b`. One example of using this attribute is to model traveling times, when jobs are executed at different locations. Another example of using this attribute is to model cleaning times of a paint machine, when the cleaning time depends on the color used during the previous job. All entries of this matrix are assumed to be 0 when not specified. If the schedule domain is a calendar, the unit of measurement is the time unit of the schedule domain; otherwise the unit of measurement is unitless. This matrix can, but need not, be symmetric. In the constraint programming literature, this attribute is also called *sequence-dependent setup times* or *changeover times*. The syntax for this attribute is as follows:

activity-transition :



An example of a transition specification is:

```
Resource myMachine {
    ScheduleDomain  :  H;
    Usage           :  sequential;
    Activities      :  acts(a), maintenance;
    Transition      : {
        (acts(a1),acts(a2))    : travelTime(a1,a2),
        (maintenance,acts(a1)) : travelTime('home',a1),
        (acts(a1),maintenance) : travelTime(a1,'home')
    }
    Comment         : {
        "activities acts are executed on location/site; yet
        maintenance is executed at home.  Transitions are
        the travel times between locations."
    }
}
```

The `Transition` attribute is not mandatory.

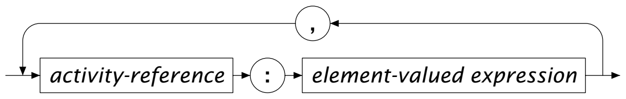
The GroupSet attribute

The GroupSet attribute is only available to sequential resources. The elements of this set name the groups into which the activities can be divided. This attribute is not mandatory.

The GroupDefinition attribute

The GroupDefinition attribute is only available when the GroupSet attribute has been specified. It contains a mapping of activities to group set elements. This mapping is essential for the GroupTransition attribute and for the intrinsic functions cp::GroupOfNext and cp::GroupOfPrevious. The syntax is as follows:

group-definition :

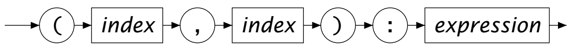


This attribute is mandatory when the GroupSet attribute has been specified.

The GroupTransition attribute

The GroupTransition attribute is used to specify the transition times/sequence dependent setup times between activities in a compressed manner. This attribute is only available when the GroupSet attribute has been specified. The syntax is:

activity-group-transition :



Consider an application where each city has to be visited by a car on multiple occasions, to bring goods being produced in one city to another city where they are consumed. The first product is consumed before the last product is produced:

```
Activity VisitCity {
    IndexDomain      :  (car,city,iter);
    ScheduleDomain   :  Timeline;
    Property         :  Optional;
}
Resource carEnRoute {
    Usage            :  sequential;
    IndexDomain      :  car;
    ScheduleDomain   :  TimeLine;
    Activities       :  VisitCity(car,city,iter);
    GroupSet         :  Cities;
    GroupDefinition  :  VisitCity(car,city,iter) : city;
    GroupTransition  :  (cityFrom,cityTo) : CityDistance(cityFrom,cityTo);
}
```

In this example, the group transition matrix is defined for each combination of cities, which is significantly smaller than an equivalent transition matrix defined for each possible combination of activities would have been. This not only saves memory, but may also save a significant amount of solution time as some Constraint Programming solvers check whether the triangular inequality holds at the start of the solution process in order to determine the most effective reasoning available to that solver. The GroupTransition attribute is not mandatory.

Property TransitionOnlyNext

The attributes Transition and GroupTransition specify the minimal time between two activities a1 and a2 if a1 *comes before* a2. By specifying the property TransitionOnlyNext, these attributes are limited to specify only the minimal distances between two activities a1 and a2 if a1 *precedes* a2. An activity a1 precedes a2, if there is no other activity b scheduled between a1 and a2. In the example that follows, a, b, and c are all activities of length 1.

```
Resource seqres {
    Usage          :  sequential;
    ScheduleDomain :  timeline;
    Activities     :  a, b, c;
    Property       :  TransitionOnlyNext;
    Transition     :  (a,b):1, (b,c):1, (a,c):7;
    Precedes       :  (a,b), (b,c);
}
```

Minimizing c.End, the solution is:

```
    a.Begin := 0 ; a.End := 1 ;
    b.Begin := 2 ; b.End := 3 ;
    c.Begin := 4 ; c.End := 5 ;
```

By omitting the TransitionOnlyNext property, the minimal distance between a and c is taken into account, and the solution becomes:

```
    a.Begin := 0 ; a.End := 1 ;
    b.Begin := 2 ; b.End := 3 ;
    c.Begin := 8 ; c.End := 9 ;
```

The Sequencing *attributes*

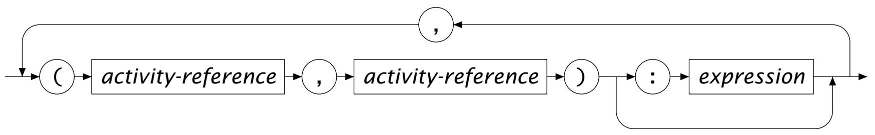
The attributes FirstActivity, LastActivity, ComesBefore, and Precedes are collectively called sequencing attributes. They are used to place restrictions on the sequence in which the activities are scheduled. These attributes are only available to sequential resources.

- **FirstActivity:** When specified, this has to be a reference to a single activity. When this activity is present, it will be the first activity in a feasible solution.
- **LastActivity:** When specified, this has to be a reference to a single activity. When this activity is present, it will be the last activity in a feasible solution.
- **ComesBefore:** This is a list of activity pairs (a,b). A pair (a,b) in this list indicates that activity a comes before activity b in a feasible solution. There may be another activity c that is scheduled between a and b in a feasible solution. This constraint is only enforced when both a and b are present.

- **`Precedes:`** This is a list of activity pairs `(a,b)`. A pair `(a,b)` in this list indicates that activity `a` precedes activity `b` in a feasible solution. There can be no other activity `c` scheduled between `a` and `b` in a feasible solution, but a gap between `a` and `b` is allowed. This constraint is only enforced when both `a` and `b` are present.

The syntax of the attributes `FirstActivity` and `LastActivity` is simply a reference to a single activity and so the syntax diagram is omitted here. The syntax diagram for the attributes `ComesBefore` and `Precedes` is more interesting:

activity-sequence :



If, following the above syntax diagram, an expression is omitted, it is taken to be 1. An example illustrating all the sequencing attributes is presented below:

```
Resource myMachine {
    ScheduleDomain  :  H;
    Usage           :  sequential;
    Activities      :  setup(a), finish(a);
    FirstActivity   :  setup('warmingUp');
    LastActivity    :  finish('Cleaning');
    ComesBefore     :  (setup(a1),setup(a2)) : taskbefore(a1,a2);
    Precedes        :  (setup(a),finish(a));
}
```

None of the sequencing attributes are mandatory.

22.2.2.2 Parallel resources

Parallel resources model and limit the resource consumption and resource production of activities that take place in parallel. Examples of parallel resources could be monetary budget and truck load.

`.ActivityLevel` suffix

A parallel resource has only one suffix, namely `.ActivityLevel`. This suffix is only affected by scheduled activities. The limits on the `.ActivityLevel` suffix, its initialization, and how it is affected by executed activities is discussed below in the parallel resource specific attributes.

The `LevelRange` attribute

The `LevelRange` attribute states the range for the activity level of a parallel resource. The maximum value represents the capacity of the resource. It cannot be specified per element in the schedule domain of the resource. The syntax of this attribute is similar to the syntax of the `Range` attribute for bounded integer variables.

```
Resource myMachine {
    IndexDomain     :  m;
    ScheduleDomain  :  h;
    Usage          :  parallel;
    Activities     :  act(a);
    LevelRange     :  {
        {minlev(m) .. maxlev(m)}
    }
}
```

The `LevelRange` attribute is only applicable for parallel resources, and for such a resource it is mandatory.

The `InitialLevel` attribute

The `InitialLevel` attribute defines the initial value of the `.ActivityLevel` suffix. if it is not specified, the `.ActivityLevel` suffix is initialized to 0. The `InitialLevel` attribute is not mandatory.

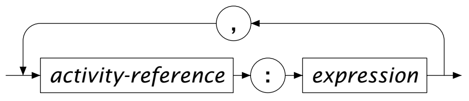
```
Resource AvailableBudget {
    ScheduleDomain  :  Hor;
    Usage           :  parallel;
    Activities      :  act(a);
    LevelRange      :  {0 .. 10000};
    InitialLevel    :  5000;
    Comment         :  "we have a starting budget of 5000";
}
```

The `.ActivityLevel` modification attributes

The attributes `LevelChange`, `BeginChange`, and `EndChange` are collectively called *`.ActivityLevel` modification* attributes.

- An activity in the `LevelChange` attribute generates a pulse: at the `.Begin` of the activity the `.ActivityLevel` of the resource is increased by the indicated amount; at the `.End` of the activity that suffix is decreased by the same amount.
- An activity in the `BeginChange` attribute increases the `.ActivityLevel` of the resource at the `.Begin` of the activity by the indicated amount.
- An activity in the `EndChange` attribute increases the `.ActivityLevel` of the resource at the `.End` of the activity by the indicated amount.

Note that not only can the indicated amount be a positive or negative integer, it can also be an integer variable. The effect of an activity on the `.ActivityLevel` is illustrated in the Figure 22.1. The syntax of these attributes is as follows:
level-modification :



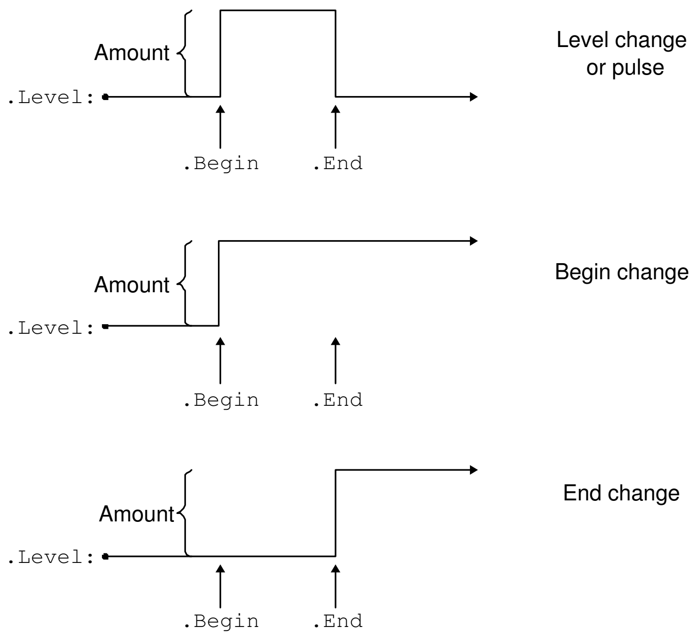


Figure 22.1: Changes to the suffix `.ActivityLevel` of a resource

The next example illustrates the use of the `.ActivityLevel` modification attributes:

```
Resource Budget {
    ScheduleDomain  :  Days;
    Usage           :  parallel;
    Activities      :  Act(i), Alt_Act(j), Deposit_Act(d);
    LevelRange      :  [0, 100];
    LevelChange     :  Alt_Act(i)    : -alt_act_budget(i);
    BeginChange     : {
        Deposit_Act(d): Deposit(d),
        Act(i)        : -ActCost(i)
    }
    EndChange       :  Act(i)        : Profit(i);
}
```

In this example, `Deposit_Act` can be modeled as an activity with a schedule domain containing only one element (an event), see Page 378. None of the `.ActivityLevel` modification attributes are mandatory, but when none of them is specified the resource is either infeasible or ineffective. When the `.Activity-Level` is outside the range of a parallel resource, that resource is infeasible.

Activity level and schedule domain

The `.ActivityLevel` suffix is not affected by holes in the schedule domain of scheduled activities. Figure 22.2 illustrates the effect of activities `A` and `B` with a level change of 1 on the resource cash. The activity `A` has its `.Begin` set to Friday, its `.End` set to Tuesday and it is not scheduled in the weekend. The activity `B` is scheduled in the weekend.

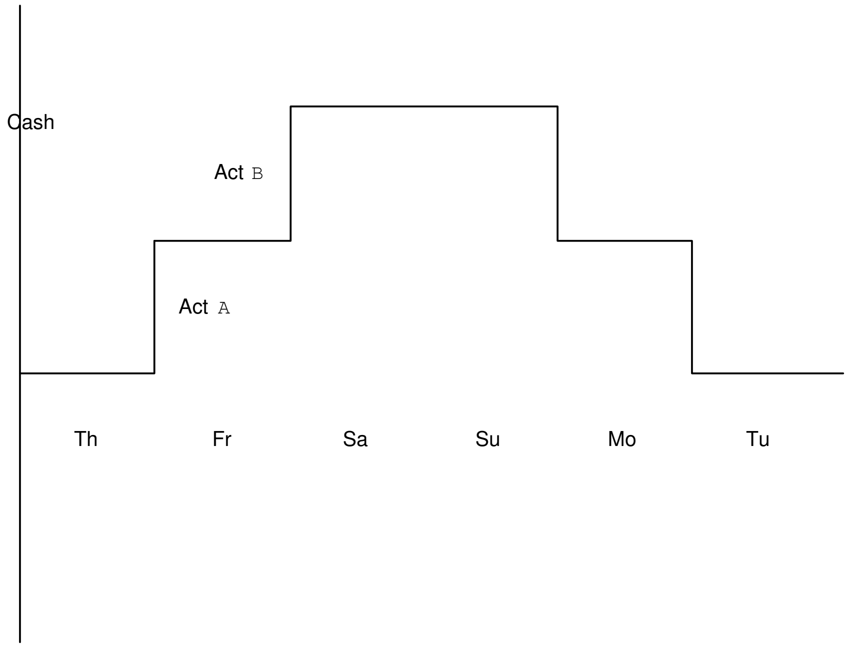


Figure 22.2: Two activities scheduled on a parallel resource

22.2.3 Functions on Activities and Scheduling constraints

Precedence constraints

The suffixes of an activity are variables, and they can be used in the formulation of constraints. Below there follows an example of a simple linear constraint which states that at least a pause of length `restTime` should be observed after activity `a` is completed before activity `b` can start.

```
a.End + restTime <= b.Begin
```

Precedence on optional activities

Consider again the inequality above, but now for optional activities `a` and `b`. When `a` is absent, the minimum value of `a.End` is meaningless but its minimum is 0 and `b` is present, this will enforce `b` to start after `restTime`. This may or may not be the intended effect of the constraint. Enforcing such constraints *only* when both activities `a` and `b` are present, the scheduling constraint `cp::EndAtStart(a,b,restTime)` can be used. This constraint is semantically equivalent to:

```
if a.Present and b.Present then
   a.End + restTime = b.Begin
endif
```

Here `restTime` is an integer valued expression that may involve variables. Note that the scheduling constraint can be exploited more effectively during the solving process than the equivalent algebraic formulation. A list of available scheduling constraints for precedence relations is given in Table 22.5.

Precedence Relations	
	When activities a and b are present and for a non-negative integer delay d
cp::BeginBeforeBegin(a,b,d)	a.Begin $+ d \le b$.Begin
cp::BeginBeforeEnd(a,b,d)	a.Begin $+ d \le b$.End
cp::EndBeforeBegin(a,b,d)	a.End $+ d \le b$.Begin
cp::EndBeforeEnd(a,b,d)	a.End $+ d \le b$.End
cp::BeginAtBegin(a,b,d)	a.Begin $+ d = b$.Begin
cp::BeginAtEnd(a,b,d)	a.Begin $+ d = b$.End
cp::EndAtBegin(a,b,d)	a.End $+ d = b$.Begin
cp::EndAtEnd(a,b,d)	a.End $+ d = b$.End
Scheduling Constraints	**Interpretation**
cp::Span(g,i,a_i)	The activity g spans the activities a_i g.Begin $= \min_i a_i$.Begin$\wedge$ g.End $= \max_i a_i$.End
cp::Alternative(g,i,a_i)	Activity g is the single selected activity a_i $\exists j : g = a_j \wedge \forall k, j \neq k : a_k$.present $= 0$
cp::Synchronize(g,i,a_i)	If g is present, all present activities a_i are scheduled at the same time. g.present $\Rightarrow (\forall i : a_i$.present $\Rightarrow g = a_i)$

Table 22.5: Constraints for scheduling

Global scheduling constraints

In addition to these precedence constraints and the constraints that are defined by resources, AIMMS offers several other global constraints that are helpful in modeling scheduling problems. Table 22.5 presents the global scheduling constraints and functions available in AIMMS. These constraints are based on activities and can be used to represent hierarchical planning problems (cp::Span), to schedule activities over alternative resources (cp::Alternative), and to synchronize the execution of multiple activities (cp::Synchronize).

Functions on activities

There are several functions available that provide control over the value to be used for the suffixes of activities in the case of absence. In addition, there are functions available for relating adjacent activities on a resource. Table 22.6 lists the functions available that operate on activities. As an example, consider a model whereby the length of two adjacent jobs is limited:

```
Set Timeline {
    Index          :  tl;
}
Set Jobs {
    Index          :  j;
}
```

Limiting activity suffixes taking absence into account	
	a is the activity
	d the absence value
`cp::ActivityBegin`(a,d)	Return begin of activity
`cp::ActivityEnd`(a,d)	Return end of activity
`cp::ActivityLength`(a,d)	Return length of activity
`cp::ActivitySize`(a,d)	Return size of activity
Adjacent Activity	
	r is the resource
	s is the scheduled activity
	e is extreme value (when s is first or last)
	a is absent value (s is not scheduled)
`cp::BeginOfNext`(r,s,e,a)	Beginning of next activity
`cp::BeginOfPrevious`(r,s,e,a)	Beginning of previous activity
`cp::EndOfNext`(r,s,e,a)	End of next activity
`cp::EndOfPrevious`(r,s,e,a)	End of previous activity
`cp::GroupOfNext`(r,s,e,a)	Group of next activity, see also page 385
`cp::GroupOfPrevious`(r,s,e,a)	Group of previous activity
`cp::LengthOfNext`(r,s,e,a)	Length of next activity
`cp::LengthOfPrevious`(r,s,e,a)	Length of previous activity
`cp::SizeOfNext`(r,s,e,a)	Size of next activity
`cp::SizeOfPrevious`(r,s,e,a)	Size of previous activity

Table 22.6: Functions for scheduling

```
Parameter JobLen {
    IndexDomain     :  j;
}
Activity doJob {
    IndexDomain     :  j;
    ScheduleDomain  :  Timeline;
    Length          :  Joblen(j);
}
Resource aWorker {
    Usage           :  sequential;
    ScheduleDomain  :  Timeline;
    Activities      :  doJob(j);
}
Constraint LimitLengthTwoAdjacentJobs {
    IndexDomain     :  j;
    Definition      : {
        cp::ActivityLength(doJob(j),0) +
        cp::LengthOfNext(aWorker,doJob(j)) <= 8
    }
}
```

In the constraint `LimitLengthTwoAdjacentJobs` we take the length of job via the function `cp::ActivityLength` and the length of the next job for resource `aWorker` via the function `cp::LengthOfNext`. In the above constraint, the use of the function `cp::ActivityLength` is not essential because activity `doJob` is not optional. We can use the suffix notation instead and the constraint becomes:

```
Constraint LimitLengthTwoAdjacentJobs {
    IndexDomain     :  j;
    Definition      : {
        doJob(j).Length +
        cp::LengthOfNext(aWorker,doJob(j)) <= 8
    }
}
```

22.2.4 Problem schedule domain

The problem schedule domain of a mathematical program is a single named set containing all timeslots referenced in the activities, resources and timeslot valued element variables of that mathematical program. For the activities to be scheduled, we are usually interested in when they take place in real time; the mapping to real time is an ingredient of the solution. An exception might be if we are interested in the process of scheduling instead of its results. In that case, a contiguous subset of the `Integers` suffices. Contiguous subsets of `Integers` are supported by AIMMS, but not considered in the remainder of this subsection.

Real world representations

The problem schedule domain represents the period of time on which activities are to be scheduled. This portion of time is discretized into timeslots. Consider the following three use cases for a problem schedule domain.

1. The first use case is probably the most common one; the distance in time between two consecutive timeslots is constant. This distance is equal to a single unit of time. As an example, consider an application that constructs a maintenance scheme for playing fields. Two consecutive line painting activities should not be scheduled too close or too far from each other. Similarly for other consecutive activities of the same type such as garbage pickup. In this use case the distance in time between two consecutive timeslots is meaningful. A `Calendar` is practical for this use case.
2. The second use case is the one in which the distance in time between two consecutive timeslots is not constant. As an example, consider an application that constructs a sequence of practice meetings for teams with a limited number of playing fields available. The set of available dates is based on the team member availability, and the distance in time between two consecutive time slots may vary. An important restriction for this application is that two meetings with the same type of practice should be some number of meetings apart. In addition, we want to avoid, for each team, peaks and big holes in exercise dates by limiting the number of exercise dates skipped between two consecutive exercises.

3. The third use case is a combination of the other two use cases. The problem schedule domain is again one whereby the distance in time between two consecutive timeslots is constant. In addition, there are subsets of this problem schedule domain which apply to selected activities and resources. As an example, consider an application that schedules both maintenance activities and team practice sessions on a set of playing fields.

The above use cases are illustrated in AIMMS below.

Use case 1: constant distance

In the first use case as illustrated by the example below, the problem schedule domain is the calendar `yearCal`. Two consecutive timeslots in that calendar have the fixed distance of 1 day. The activity `FieldMaintenance` models that there are various types of maintenance activities to be scheduled for the various fields, and each type of maintenance may occur more than once. The two constraints in this example restrict the minimal and maximal distance between consecutive maintenance activities of the same type on the same field.

```
Calendar yearCal {
    Index           :  d;
    Unit            :  day;
    BeginDate       :  "2012-01-01";
    EndDate         :  "2012-12-31";
    TimeslotFormat  :  "%c%y-%sm-%sd";
}
Activity FieldMaintenance {
    IndexDomain     :  (pf, mt, occ);
    ScheduleDomain  :  yearCal;
    Property        :  Optional;
    Length          :  1[day];
    Comment         : {
        "Maintenance on
        playing field     pf
        maintenance type  mt
        occurrence       occ"
    }
}
Constraint maintenanceMinimalDistance {
    IndexDomain     :  (pf, mt, occ) | occ <> first( occurrences );
    Text            :  "at least 7 days apart.";
    Definition      : {
        cp::BeginBeforeBegin( FieldMaintenance(pf, mt, occ-1),
        FieldMaintenance(pf, mt, occ), 7 )
    }
}
Constraint maintenanceMaximalDistance {
    IndexDomain     :  (pf, mt, occ) | occ <> first( occurrences );
    Text            :  "at most 14 days apart.";
    Definition      : {
        cp::BeginBeforeBegin( FieldMaintenance(pf, mt, occ),
        FieldMaintenance(pf, mt, occ-1), -14 )
    }
}
```

Use case 2: varying distance

For the second use case, the same calendar `yearCal` is declared as in the first use case, in order to relate to real time. We are interested in only a selection of the dates available, and a subset `exerciseCal` is created from this calendar. In order to remove the fixed distance from the timeslots, the elements in `exerciseCal` are copied to `exerciseDates`.

```
Calendar yearCal {
    ...
}
Set  exerciseCal {
    SubsetOf       :  yearCal;
    Index          :  yc;
}
Set exerciseDates {
    Index          :  ed;
    Comment        :  "Constructed in ...";
}
```

A simple way of copying the elements with their names from `exerciseCal` to `exerciseDates` is in the code fragment below.

```
        Empty exerciseDates ;
        for yc do
            exerciseDates += StringToElement(exerciseDates,
             formatString("%e",yc), create:1 );
        endfor ;
```

Now that we have the set of exercise dates without a time unit associated, we can use it to declare exercise activities for each team and enforce a minimal distance between exercises of the same type as illustrated below. This minimal distance will now be enforced using a sequential resource and counting the number of times a particular exercise type occurred.

```
Set exerciseTypes {
    Index          :  et;
}
Activity gExerciseTeam {
    IndexDomain     :  (tm,occ);
    ScheduleDomain  :  exerciseDates;
    Length          :  1;
    Comment         : {
        "occ in Occurrencs, defining the number of times
        team tm has to exercise"
    }
}
Resource teamExercises {
    Usage           :  sequential;
    IndexDomain     :  tm;
    ScheduleDomain  :  exerciseDates;
    Activities      :  gExerciseTeam(tm, occ);
    Precedes        :  (gExerciseTeam(tm, occ1),gExerciseTeam(tm, occ2)):occ1=occ2-1;
    Comment         :  "Purpose: determine when a team may exercise";
}
Activity exerciseTeam {
    IndexDomain     :  (tm,et,occ);
    ScheduleDomain  :  exerciseDates;
    Property        :  optional;
    Length          :  1;
}
```

```
Constraint oneExerciseType {
    IndexDomain     :  (tm,occ);
    Definition      : {
        cp::Alternative(
        globalActivity  :  gExerciseTeam(tm, occ),
        activityBinding :  et,
        subActivity     :  ExerciseTeam(tm, et, occ))
    }
    Comment         :  "Purpose: select a single type of exercise";
}
Constraint doingAnExerciseTypeAtMostOnceOverOccurrences {
    IndexDomain     :  (tm,et,occ) | occ <> first( occurrences );
    Definition      : {
        sum( occ1 | occ1 <= occ and occ1 < occ + 2,
        ExerciseTeam(tm, et, occ).Present ) <= 1
    }
    Comment         : {
        "Purpose: the same type of exercise should be some
        exercises apart"
    }
}
Constraint avoidSmallHolesBetweenExercises {
    IndexDomain     :  (tm,occ) | occ <> first( occurrences );
    Text            :  "at least minHoleSize exercise dates apart.";
    Definition      : {
        cp::EndBeforeBegin( gExerciseTeam(tm, occ-1),
        gExerciseTeam(tm, occ), minHoleSize )
    }
}
Constraint avoidBigHolesBetweenExercises {
    IndexDomain    :  (tm,occ) | occ <> first( occurrences );
    Text            :  "at most maxHoleSize exercise dates apart.":
    Definition      : {
        cp::BeginBeforeEnd( gExerciseTeam(tm, occ),
        gExerciseTeam(tm, occ-1), -maxHoleSize )
    }
}
```

Use case 3: combination

There can be only one problem schedule domain per mathematical program; we use the one from the first use case as it encompasses the one from the second use case. Thus we need to adapt the schedule domains of selected activities and resources to the following:

```
Activity gExerciseTeam {
    IndexDomain     :  (tm,occ);
    ScheduleDomain  :  exerciseCal ! modified;
    Length          :  1[day]      ! modified;
}
Resource OneExercise {
    Usage           :  sequential;
    IndexDomain     :  tm;
    ScheduleDomain  :  exerciseCal ! modified;
    Activities      :  gExerciseTeam(tm, occ);
    Precedes        :  (gExerciseTeam(tm, occ1),gExerciseTeam(tm, occ2)):occ1=occ2-1;
    Comment         :  "Purpose: determine when a team may exercise";
}
Activity exerciseTeam {
    IndexDomain    :  (tm,et,occ);
    ScheduleDomain  :  exerciseCal ! modified;
    Property       :  optional;
    Length         :  1[day]      ! modified;
}
```

The constraints `oneExerciseType` and `doingAnExerciseTypeAtMostOnceOverOccurrences` can be left unchanged as they are not dependent on the distance between timeslots. The constraints `avoidSmallHolesBetweenExercises` and `avoidBigHolesBetweenExercises` is dependent on the distance between elements of `exerciseDates` and will now have to be remodeled. By using the fact that the `.Size` refers to the number of elements in the schedule domain of an activity between its `.Begin` and `.End` and that we can define an activity that encompasses a hole by spanning the previous exercise and the current one, the remodeling is done as follows:

```
Activity encompassHoleBetweenExercises {
    IndexDomain     :  (tm,occ)|occ <> first(occurrences);
    ScheduleDomain  :  exerciseCal;
}
Constraint defineEncompassHoleBetweenExercises {
    IndexDomain     :  (tm,occ)|occ <> first(Occurrences);
    Definition      : {
        cp::Span(
        globalActivity  :  encompassHoleBetweenExercises(tm, occ),
        activityBinding :  occ1 | occ1 = occ or occ1 = occ-1,
        subActivity     :  gExerciseTeam(tm, occ1))
    }
}
Constraint maxSizeEncompassingActivity {
    IndexDomain     :  (tm,occ)|occ <> first(Occurrences);
    Definition      : {
        encompassHoleBetweenExercises(tm,occ).size <=
        (maxHoleSize + 2)[day]
    }
}
Constraint minSizeEncompassingActivity {
    IndexDomain    :  (tm,occ)|occ <> first(Occurrences);
    Definition     : {
        encompassHoleBetweenExercises(tm,occ).size >=
        (minHoleSize + 2)[day]
    }
}
```

comparing use cases 2 and 3

Only when the distance in real time is not relevant to an application, the representation chosen for use case 2 has the following advantages over the representation chosen for use case 3:

- The limiting of the size of a hole is formulated more directly using the `cp::EndBeforeBegin` and `cp::BeginBeforeEnd` than one using an additional activity leading to a formulation that is easier to understand.
- We not only lose the power of propagation provided by the global constraints `cp::EndBeforeBegin` and `cp::BeginBeforeEnd` in use case 3, but we also introduce another activity, and thus additional variables, which increases the search space thereby decreasing the perfomance further.

- The set exerciseDates is smaller than the set yearCal. It is generally a good idea to keep the sets used as ranges for element variables small including the schedule domain to be considered.

This concludes the discussion on multiple use cases for problem schedule domains.

The attribute ScheduleDomain of a mathematical program

When an application contains activities, the attribute ScheduleDomain becomes available to a mathematical program declared in that application. Unlike the attribute with the same name for activities and resources, only a single named one-dimensional set can be filled in here. If this attribute is filled in, with, say Timeline, then AIMMS will ensure that each activity and resource has a schedule domain that is a subset of Timeline. In addition, for each element variable with a range say, selectedMoments, where Timeline and selectedMoments have the same root set, AIMMS will verify that selectedMoments is a subset of Timeline. In addition, the problem schedule domain, say Timeline, has to meet one of the following three requirements:

- Timeline is a true subset of the set Integers. In order to make references forward and backward in time unambiguous, it is required that Timeline is contiguous.
- Timeline is a subset of a calendar. Again, in order to make references forward and backward in time unambiguous, it is required that Timeline is contiguous.
- Timeline is not a subset of Integers nor a subset of a calendar. No additional requirements are placed on Timeline.

This attribute is not mandatory.

Deriving the problem schedule domain

If this attribute is not filled in, then AIMMS will derive the problem schedule domain by finding the smallest common superset of the schedule domains of the activities and resources of the mathematical program. If this set is not a calendar, but a subset thereof, AIMMS chooses to use the calendar instead. This is motivated by the assumption that the length between timeslots is usually relevant in scheduling applications. In scheduling applications in which the subset is more appropriate, copying the elements of a calendar provides an alternative, as illustrated above.

22.3 Modeling, solving and searching

Introduction

In this section we explain how constraint programming models are formulated and solved in AIMMS. We start by explaining how constraint programming formulations are fit into the paradigms of AIMMS like the free objective and units of measurement. We then explain the different mathematical programming types associated to constraint programming, and finally we discuss how a user can modify the search procedure in AIMMS.

The free objective

By design, the objective of any mathematical program in AIMMS is a *free* variable, even when it can be deduced that the objective function is always integer valued for a feasible solution. However, for an infeasible solution, AIMMS will assign the special value NA to the objective variable, in order to emphasize that a feasible solution is not available. Having a single variable for the objective function is convenient when communicating its value in the progress window and other places of AIMMS.

22.3.1 Constraint programming and units of measurement

Integer coefficients only

Constraint programming solvers require the coefficients used in constraints to be integer; fractional values or special values such as -inf, zero, na, undf, or inf are not allowed. In applications where the choice of base units is free, fractional values are easily avoided by choosing the base unit of quantities, and the units of variables and constraints such that all amounts to be considered are integer multiples of those base units, as detailed in Section 32.5.1. As an example, consider a simple constraint stating that the integer variable y is 0.9[m] away from the integer variable x. Both variables are multiples of the derived unit dm, i.e., 0.1[m]. The variables and constraints are declared as follows:

```
Variable x {
    Range      :  integer;
    Unit       :  dm;
}
Variable y {
    Range      :  integer;
    Unit       :  dm;
}
Constraint y_away_from_x {
    Unit       :  dm;
    Definition :  y = abs( x - 0.9[m] );
}
```

Using the unit m as a base unit, this will lead to fractional values, as can be seen in the constraint listing:

```
y_away_from_x .. [ 1 | 1 | before ]

    y =
    abs((x-0.9))  ****

    name            lower          level           upper           scale
    x                   0              0 21474836470.000           0.100
    y                   0              0 21474836470.000           0.100
```

The coefficient for distance, 0.9[m], is in meters and the variables x and y have the same units inside the row. This row is scaled back to dm afterwards. As a result of all this scaling, the computations go from the domain of integer arithmetic into the domain of floating point arithmetic. For constraint programming, we need to avoid the domain of floating point arithmetic.

Adapted base unit

We will continue the above example, by adapting the base unit such that all amounts are integral multiples of that base unit; we select dm as a base unit. This will lead to the following row communicated to the solver:

```
y_away_from_x .. [ 1 | 1 | before ]

    y =
    abs((x-9))  ****

    name       lower      level      upper
    x              0          0 2147483647
    y              0          0 2147483647
```

Observe that scaling is not needed anymore. Using the scaling based on dm instead of m will keep all computations during the solution process in the domain of integer arithmetic.

Quantity based scaling

In the example of the previous paragraph, the base unit was adapted to the needs of constraint programming; namely to stay within the domain of integer arithmetic. For multi-purpose applications, freedom of choosing the base units of quantities according to the needs of a constraint programming problem is not always available. In order to stay within the domain of integer arithmetic, we can associate a convention with the mathematical program and filling in the per quantity attribute, see also Section 32.8. By filling in the per quantity attribute, AIMMS will generate the mathematical program where all coefficients are scaled with respect to the units specified in the per quantity attribute. Let us continue the example of the previous paragraph using m as the base unit and adding a convention to the mathematical program.

```
Quantity SI_Length {
    BaseUnit     :  m;
    Conversions  : {
        dm -> m : # -> # / 10,
        cm -> m : # -> # / 100
    }
    Comment      :  "Expresses the value of a distance.";
}
Parameter LengthGranul {
    InitialData  :  10;
}
Convention solveConv {
    PerQuantity  :  SI_Length : LengthGranul * cm;
}
MathematicalProgram myCP {
    Direction    :  minimize;
    Constraints  :  AllConstraints;
    Variables    :  AllVariables;
    Type         :  Automatic;
    Convention   :  solveConv;
}
```

Again, AIMMS will generate the constraint such that only integer arithmetic is needed. The constraint listing of that constraint is similar to the constraint listing presented in the paragraph *adapted base unit* above and not repeated.

Note also that with conventions, we can now use parameters to further control the scaling; if we want to change the model such that we can use multiples of `20[cm]` instead of multiples of `10[cm]`, we only need to change the value of `LengthGranul`.

Calendar used for timeline

Scheduling applications in which the schedule domain is based on a calendar, the length of a timeslot is equal to the unit of the calendar, see Section 33.2. The time quantity is overridden, as if an entry in the `per quantity` attribute of the associated convention is given, selecting the calendar unit. Even if no convention was associated with the mathematical program. In short, for scheduling applications, AIMMS will scale time based data according to the length of a timeslot.

22.3.2 Solving a constraint program

Mathematical Programming types

AIMMS distinguishes two types of mathematical programs that are associated with constraint programming models: `COP` for constraint optimization problems, and `CSP` for constraint satisfaction problems. Both `COP` and `CSP` are exact in that `COP` provides a proven optimal solution while `CSP` provides a solution, or proves that none exist, if time permits.

Limiting solution time

Constraint programming problems are combinatorial problems and therefore may take a long time to solve, especially when trying to prove optimality. In order to avoid unexpectedly long solution times, you can limit the amount of time allocated to the solver for solving your problem as follows:

```
solve myCOP where time_limit := pMaxSolutionTime ;
! pMaxSolutionTime is in seconds.
```

Alternatively, when you are satisfied with the current objective, as presented in the progress window, and not want to wait on further improvements, you can interrupt the solution process by using the key-stroke *ctrl-shift-s*.

22.3.3 Search Heuristics

Search heuristics

During the solving process, constraint programming employs search heuristics that define the shape of the search tree, and the order in which the search tree nodes are visited. The shape of the search tree is typically defined by the order of the variables to branch on, and the corresponding value assignment. AIMMS allows the user to specify which variable and value selection heuristics are to be used. For example, to decide the next variable on which to branch, a commonly used search heuristic is to choose a non-fixed variable with the minimum domain size, and assign its minimum domain value.

Search phases

The first method offered by AIMMS to influence the search process is through using the `Priority` attributes of the variables. AIMMS will group together all variables that have the same priority value, and each block of variables will define a *search phase*. That is, the solver will first assign the variables in the block with the highest priority, then choose the next block, and so on. As discussed in Section 14.1.1, the highest priority is the one with the highest positive value. Defining search phases can be very useful. For example, when scheduling activities to various alternative resources, it is natural to first assign an activity to its resource before assigning its begin.

Variable and value selection

The variable and value selection heuristics offered by AIMMS are presented in Table 22.7. They can be accessed through the 'solver options' configuration window. As an example, we can define a 'constructive' scheduling heuristic that builds up the schedule from the begin of the schedule domain by using `MinValue` as the variable selection, and `Min` as the value selection. Indeed, this heuristic will attempt to greedily schedule the activities as early as possible. Note that both these variable and value heuristics apply to the entire search process. If no variable priorities are specified, the variable selection heuristic will consider all variables at a time. Otherwise, the variable selection heuristic is applied to each block individually.

Heuristic	**Interpretation**
Variable selection:	choose the non-fixed variable with:
`Automatic`	use the solver's default heuristic
`MinSize`	the smallest domain size
`MaxSize`	the largest domain size
`MinValue`	the smallest domain value
`MaxValue`	the largest domain value
Value selection:	assign:
`Automatic`	use the solver's default heuristic
`Min`	the smallest domain value
`Max`	the largest domain value
`Random`	a uniform-random domain value

Table 22.7: Search heuristics

Bibliography

[Ba01] P. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-based scheduling*, Kluwer Academic Publishers, 2001.

[Ro06] F. Rossi, P. Van Beek, and editors T. Walsh, *Handbook of constraint programming*, Elsevier, 2006.