Developing Constraint Programming Applications with AIMMS

Willem-Jan van Hoeve

Tepper School of Business, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
vanhoeve@andrew.cmu.edu

Abstract. We describe the constraint programming interface of the optimization modeling systems Aimms. First, we present the modeling language for basic constraint programming and advanced scheduling constructs, and specify how search can be controlled. Then we provide three example applications that illustrate how Aimms can be used for developing constraint programming applications.

1 Introduction

Over the last decades, constraint programming (CP) has proved to be an important tool for solving combinatorial optimization problems, either as a stand-alone technology [17], or in combination with other optimization methods such as integer programming or local search [13, 7, 14]. In addition to specific solving methodology (i.e., systematic search combined with logic-based inference methods), CP offers a wealth of modeling constructs beyond the classical (non)linear inequalities that are used in (non)linear programming, such as logical operators and variable-ranged indices.

The need for algebraic modeling languages, such as AMPL, GAMS, and AIMMS, to embrace CP technology was already acknowledged and motivated by Fourer and Gay [4]. For example, they argue that CP can provide more natural formulations that are closer to the original problem than more restricted languages such as integer linear programming. Furthermore, CP technology can be a very effective solving tool, especially for highly combinatorial problems such as complex scheduling applications. Lastly, generic algebraic modeling languages (as opposed to vendor-specific systems) allow to interface with a variety of solvers. Therefore by adding CP, modeling systems allow to express a wider range of combinatorial problems, and to solve those with a broader set of solution methods.

In this work, we focus on the constraint programming interface of Aimms. The modeling language underlying Aimms was originally developed in a similar spirit as GAMS [2] and AMPL [5]. Similar to those systems, it is based on an algebraic syntax and offers access to (at least) integer linear programming (ILP), quadratic programming (QP), and nonlinear programming (NLP) technology. The main difference with these other languages, however, is that model development in Aimms revolves around a graphical modeling interface depicting the hierarchical structure in a model formulation. This allows to formulate a problem in an intuitive and naturally decomposed manner. A second important feature of AIMMS is that it offers user-developed 'pages', that can be used to graphically depict solutions or even build entire end-user applications. In fact, an important benefit of AIMMS is that it can not only be used to quickly develop a prototype, but also subsequently easily enhance that prototype into a user friendly end-user application through graphical pages, or an application deployed as part of the software framework of a company.

The goal of this paper is twofold. We first give an overview of the constraint programming interface of AIMMS. We then illustrate how this interface can be used to develop constraint programming applications, by providing three examples. The first is a map coloring problem that illustrates the GIS support for visualization. The second is a column generation procedure that illustrates the ease with which hybrid solution methods, in this case combining LP and CP, can be implemented. The third is real-world application for inventory balancing in bike sharing systems.

We note that a description of the implementation details of the constraint programming interface of AIMMS will be presented by Kuip [9] in the CP 2013 workshop on "CP Solvers: Modeling, Applications, Integration, and Standardization".

2 Related Work

Several other industrial and academic modeling languages/systems have been developed to make the use of CP and hybrid methods more accessible. These include IBM ILOG CPLEX Optimization Studio (with the OPL language [20]), Fico Xpress Optimization Studio (with the Mosel language [3]), Comet [21], Zinc [11], Essence [6], AMPL [4], and SIMPL [22]. Each of these systems provides considerable advances in the usability of CP including the usability of CP by non-experts. The recently introduced CP extension of AIMMS contributes to those developments by focusing at an industrial-strength, solver-independent, intuitive graphical modeling interface. Specific differences with existing systems are that AIMMS supports a detailed interface for scheduling problems, based on the intuitive activity/resource view. In addition, AIMMS supports set based type checking on the arguments of parameters and variables when variables are used in the indexing. Finally, AIMMS offers if-then-else expressions where variables are allowed in the condition.

OPL, Comet and Mosel do offer support for developing GUI's around applications but are linked to vendor specific solvers. In addition, these three systems offer no or limited support for the nonlinear mathematical programming model types. AMPL and Zinc are not linked to specific solvers but they do not provide support for developing GUI's. Furthermore, Zinc and Essence do not offer support for the nonlinear mathematical programming modeling classes. Closest to the AIMMS interface is the constraint programming extension of AMPL [4]. However, several concepts proposed in [4] are not yet realized in AMPL, for example the use of variables as subscript in `element` constraints. SIMPL is a separate development targeted at developing applications by combining existing templates for portions of frequently occurring sub models. It does not offer a general algebraic syntax for describing arbitrary models.

There are also features in which AIMMS is not as developed as some of the systems mentioned above. For instance, set-valued variables are offered by OPL and Zinc and not yet implemented in AIMMS. In addition, the languages OPL, Zinc and Mosel offer more search directives than AIMMS does.

3 Constraint Programming Interface

Here we provide a brief overview of the AIMMS system for specifying CP applications. We provide the basic CP constructs, the interface for advanced scheduling applications, and the search specification. More details can be found in [16].

3.1 Basic Constraint Programming Constructs

Essential to a language supporting constraint programming concepts is its representation of discrete variables, the constraints that can be formulated using these variables and the available global (or symbolic) constraints.

Variable types The existing non-CP variable types of AIMMS are integer variables and continuous variables, which can be used to model, e.g., MIP and (MI)NLP problems. The integer variable type is also available to formulate CP models. In addition, the new variable type *element variable* has been introduced for CP. Analogous to the existing 'element parameter' type in AIMMS, these variables can be used to represent subscripts in an array (historically modeled through `element` constraints in CP). Other examples of their application include usage in table constraints and comparison with other element variables. Element variables can only be used in CP models. As an illustration, consider the following partial model for a warehouse location problem, in which we are given a set of clients, indexed by `c`, that must be supplied by a set of warehouses:

```
ELEMENT VARIABLE:
   identifier   :  supplyingWarehouse
   index domain :  c
   range        :  Warehouses

VARIABLE:
   identifier   :  TotalCost
   range        :  Integer
   definition   :  sum(c, SupplyCost(c,supplyingWarehouse(c)))
```

Here `supplyingWarehouse(c)` is an element valued expression resulting in an element of the set `Warehouses`. We note that this enables to detect, at compile time, the error in the expression `SupplyCost(supplyingWarehouse(c),c)`.

`cp::AllDifferent`
`cp::BinPacking`
`cp::Cardinality`
`cp::Channel`
`cp::Count`
`cp::Lexicographic`
`cp::ParallelSchedule`
`cp::Sequence`
`cp::SequentialSchedule`

(a) Combinatorial constraints

`cp::ActivityBegin`	`cp::EndBeforeEnd`
`cp::ActivityEnd`	`cp::EndOfNext`
`cp::ActivityLength`	`cp::EndOfPrevious`
`cp::ActivitySize`	`cp::GroupOfNext`
`cp::BeginAtBegin`	`cp::GroupOfPrevious`
`cp::BeginAtEnd`	`cp::LengthOfNext`
`cp::BeginBeforeBegin`	`cp::LengthOfPrevious`
`cp::BeginBeforeEnd`	`cp::SizeOfNext`
`cp::BeginOfNext`	`cp::SizeOfPrevious`
`cp::BeginOfPrevious`	
`cp::EndAtBegin`	`cp::Alternative`
`cp::EndAtEnd`	`cp::Span`
`cp::EndBeforeBegin`	`cp::Synchronize`

(b) Advanced scheduling constraints

Fig. 1. Global constraints supported by Aimms.

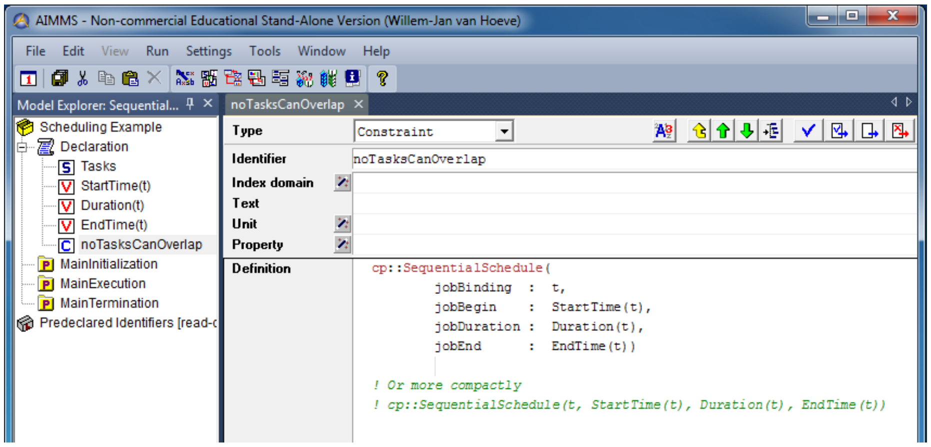
Basic constraints Interestingly, many modeling concepts from CP were already present in the existing Aimms syntax, albeit restricted to non-variable identifiers. Namely, Aimms already offers all standard arithmetic, logical, and set related operators. For MIP models, these functions can for example be applied to condition the index set over which a constraint is defined. For use in CP only a semantic change was made to the language, allowing variables to appear in these expressions. The constraints thus formed can then be added to a CP model. As an illustration, we can model a `table` constraint (which explicitly represents the set of allowed tuples for a set of variables) as follows:

```
CONSTRAINT:
   identifier : MyTableConstraint
   definition : (Var1, Var2, Var3) in MyThreeDimRelation
```

Here, `MyThreeDimRelation` represents the set of allowed tuples, and the set operator `in` defines the actual constraint.

Naturally, the CP interface of Aimms allows to use the existing operators to build any algebraic or logical expression, as is common in CP.

Global constraints Aimms offers several global constraints, as indicated in Fig. 1(a). Most of these global constraints will be familiar to CP users, as they appear under the same name in the literature. There are two new names in this list: `ParallelSchedule` and `SequentialSchedule`, corresponding to the classical 'cumulative resource' and 'unary resource' constraints of CP. These names were chosen because they are more intuitive to non-experts than the names existing in the literature for these constraints, i.e., `cumulative` and `unary` or `disjunctive`. An illustration of `SequentialSchedule` in Aimms is presented in Fig. 2.


AIMMS - Non-commercial Educational Stand-Alone Version (Willem-Jan van Hoeve)
File Edit View Run Settings Tools Window Help
Model Explorer: Sequential...
noTasksCanOverlap
Scheduling Example
Declaration
Tasks
StartTime(t)
Duration(t)
EndTime(t)
noTasksCanOverlap
MainInitialization
MainExecution
MainTermination
Predeclared Identifiers [read-c

Type: Constraint
Identifier: noTasksCanOverlap
Index domain:
Text:
Unit:
Property:
Definition:

```
cp::SequentialSchedule(
        jobBinding  :  t,
        jobBegin    :  StartTime(t),
        jobDuration :  Duration(t),
        jobEnd      :  EndTime(t))

! Or more compactly
! cp::SequentialSchedule(t, StartTime(t), Duration(t), EndTime(t))
```

Fig. 2. Using the global constraint `cp::SequentialSchedule` in AIMMS. Here, `t` is the index representing the set of tasks, `StartTime(t)`, `Duration(t)`, `EndTime(t)` are variables representing the start time, duration, and end time of each task. The argument roles such as `jobBinding` were entered by pressing ctrl-shift-space twice half-way typing `cp::SequentialSchedule`.

3.2 Representing Scheduling Problems

In order to exploit the full power of CP for more complex scheduling problems, these must be represented in a specific format that can be recognized by the solver. For this, AIMMS uses a combination of the well-known concepts of 'activities' and 'resources' that are traditionally used in constraint-based scheduling [1], and special global constraints as introduced recently for IBM ILOG CP Optimizer [10]. Activities are the objects to be scheduled, and their execution will impact one or more resources.

Activities With each activity `Act`, AIMMS automatically associates variables to the components of such an activity such as variables `Act.Begin`, `Act.End`, `Act.Length`, `Act.Size` and `Act.Present`. These correspond to the begin, end, length, size, and presence of `Act`, respectively, and they can be used as normal variables throughout the entire CP model. Length and Size are two distinct concepts. The `Act.Length` is the difference between `Act.End` and `Act.Begin`. The `Act.Size` is the number of time slots in the schedule domain of `Act` between `Act.End` (exclusive) and `Act.Begin` (inclusive), i.e. the amount of time actively spent on `Act`. The effect of an activity on the resources will be modeled at the resource level, as discussed below. Activities have a mandatory attribute `schedule domain`, representing the possible dates the activity can be executed.

Resources A resource can be declared in two ways: `Sequential` or `Parallel`. The `Sequential` usage defines a disjunctive (or unary) resource, similar to the global constraint `SequentialSchedule` defined above. The `Parallel` usage defines a cumulative resource, similar to the global constraint `ParallelSchedule` defined above. A resource has the mandatory attribute `schedule domain` which is the set of timeslots that resource is available. The topmost superset of this set should be equal to the topmost superset of the schedule domains of the associated activities.

A sequential resource has the following optional sequencing attributes: `FIRST ACTIVITY`, `LAST ACTIVITY`, `COMES BEFORE`, `PRECEDES`, and `TRANSITION`, representing standard sequencing requirements, and transition times between pairs of activities.

A parallel resource has the following optional sequencing attributes: `LEVEL RANGE`, `INITIAL LEVEL`, `LEVEL CHANGE`, `BEGIN CHANGE`, and `END CHANGE`. The first two attributes specify the bounds and starting value on the resource level. The latter three attributes are defined with respect to the activities associated with the resource: each activity should impact the resource level either by a `LEVEL CHANGE` (the level is impacted during the length of the activity), or separately declaring a `BEGIN CHANGE`, `END CHANGE`, or both. In the `BEGIN CHANGE` and `END CHANGE` attributes also variables can be referenced. The use of parallel resources is illustrated in the following example, in which the activities `Act` are defined on a set indexed by `a`, while `DepositAct` is defined on a set indexed by `d`. `AmountDeposited(d)`, `-ActCost(a)` and `Profit(a)` are parameters:

```
RESOURCE:
   identifier      : Budget
   schedule domain : DaysPlannedFor
   usage           : Parallel
   activities      : Act(a), DepositAct(d)
   level range     : {0..100}
   begin change    : DepositAct(d) : AmountDeposited(d),
                   : Act(a)        : -ActCost(a)
   end change      : Act(a)        : Profit(a)
```

Resources offer the basic building block for representing scheduling problems. In addition, Aimms allows to apply several other global constraints directly to the activities, similar to those presented in [10]; see Fig. 1(b).

Discussion The scheduling interface was designed with the goal of being as intuitive as possible for any OR practitioner, while at the same time offering enough level of detail to exploit the algorithmic power. We believe that the representation in terms of activities and resources is more appealing to non-experts than using the elementary concepts of interval variables and cumul functions of IBM ILOG CP Optimizer or OPL [10].

3.3 Search

Because constraint programming is often used for solving challenging combinatorial optimization problems, problem specific search heuristics may be critical for the successful application of CP. In AIMMS, the search can be controlled by defining a (fixed) variable ordering, and by setting solver parameters.

The variable ordering can be specified via the `Priority` of the variables (a positive integer value). Variables with a smaller priority value will be considered first in the branching process. Variables of the same priority are subject to the solver's variable selection.

The search parameters that can be set in AIMMS are solver dependent, but these normally include at least a variable selection and value selection heuristic. In addition, the settings for IBM ILOG CP Optimizer include the search type (automatic, depth-first, restart, multi-point) and restart parameters (failure limit, growth factor).

Discussion Modeling systems for constraint programming often allow the explicit specification of a search procedure within the language. However, Puget [15] argued that this limits the use of CP to a small set of experts, and instead black-box solvers with automated search procedure are needed for the broad applicability of CP. This contrast between 'model and run' and 'model and search' was more recently discussed by Michel [12], who argues for hybrid systems that offer automated search but retain the flexibility of CP to specify the search, if needed.

In order to appeal to operations research practitioners, AIMMS indeed offers a 'model and run' approach. For most applications, setting the priority level of the variables is the single most important decision to influence the search behavior and obtain useful results. Offering language support for more sophisticated search specifications would require a substantial development effort, while only few users would benefit from this feature. Ultimately, in order to widen the applicability of CP technology, a model and run approach using black-box CP solvers seems unavoidable. As a consequence, it also seems unavoidable that search be automated even more, and therefore search declarations within the modeling language would become less important.

4 Applications

4.1 GIS Support for Visualization

We first illustrate the visualization support of AIMMS, on a map coloring problem. We are given the set of states of the USA, together with an adjacency list, except for the states Alaska and Hawai. The goal is to assign a color to each state such that no two adjacent states are assigned the same color.

To represent this problem we use as index sets `USAStates`, indexed by `s, s1, s2`, and SelectedColors, containing four elements. The adjacency list for a state `s` is given by `AdjacentStates(s)`. Then we model the problem as follows:

```
ELEMENT VARIABLE:
   identifier   :  stateColor
```

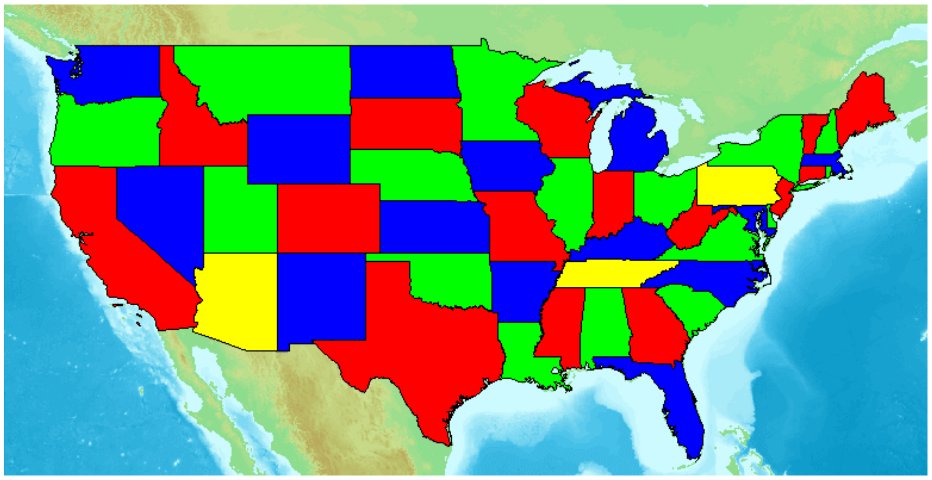


Fig. 3. Map coloring applied to the states of USA

```
    index domain :  s
    range        :  SelectedColors

  CONSTRAINT:
    identifier   :  AdjacentStatesHaveDifferentColors
    index domain :  (s1,s2)| s1 < s2 and
                             s2 in AdjacentStates(s1)
    definition   :  stateColor(s1) <> stateColor(s2)
```

Since AIMMS supports GIS technology[1] for visualization, we can use this to display our solution. Using the AIMMS 'network' object with a GIS background, the result can be displayed as in Figure 3. A detailed description of this application, together with the AIMMS model, can be downloaded from `http://blog.aimms.com/2012/12/coloring-the-states-of-the-usa/`.

4.2 Column Generation for Vehicle Routing

The next example is a column generation procedure for capacitated vehicle routing. The problem is to serve a set of clients C from a given depot with multiple trucks. Each client $c \in C$ has a given (unsplittable) load l_c that must be delivered from the depot by a truck. The total load to be picked up by a truck must not exceed the truck capacity Q. The goal is to serve all clients with minimum total travel distance.

The column generation procedure consists of two models, the master problem and the subproblem. The master problem is defined on a set `Routes` of feasible

[1] GIS stands for Geographic Information System.

truck routes. For each route r, binary parameter `ClientInRoute(r,c)` specifies whether client c is visited by r. Parameter `LengthOfRoute(r)` specifies the length of r. The master problem selects a subset of routes such that all clients are visited, with minimum total length:

```
   VARIABLE:
      identifier   :  SelectRoute
      index domain :  r
      range        :  binary

   CONSTRAINT:
      identifier   :  ClientServed
      index domain :  c
      property     :  ShadowPrice
      definition   :  sum(r, SelectRoute(r)*ClientInRoute(c, r)) >= 1

   VARIABLE:
      identifier   :  TotalDistance
      range        :  free
      definition   :  sum(r, LengthOfRoute(r)*SelectRoute(r))
```

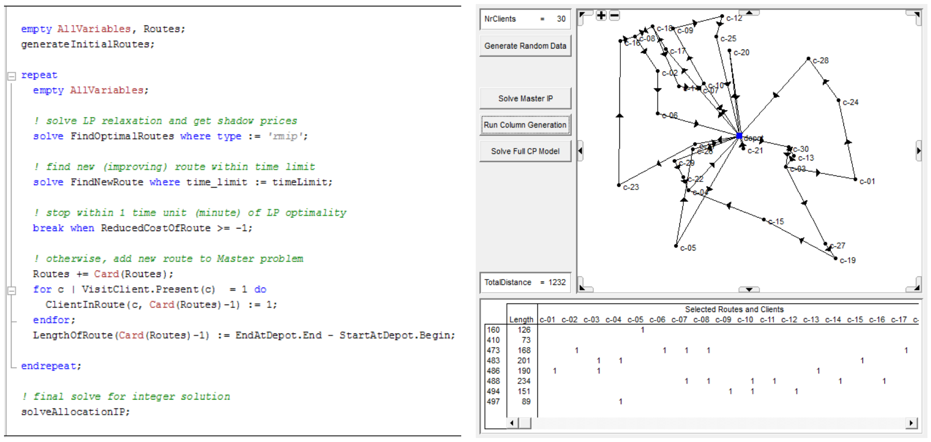
This model can be solved either as an integer program, or as a continuous linear programming relaxation. The latter provides us with a shadow price for each constraint.

In the subproblem, the goal is to find an improving truck route, relative to the shadow prices of the current LP relaxation. That is, we need to select a set of clients to visit, such that the sum of the shadow prices of these clients outweighs the total length of the route to visit these clients. In other words, the reduced cost of the new route must be negative.

This subproblem can be modeled as a CP scheduling problem, based on activities and resources [8, 18]. That is, for each client c we introduce an optional activity `ClientVisit(c)`. We moreover introduce activities `StartAtDepot` and `EndAtDepot` representing the start and end of the route. Lastly, we introduce a sequential resource `Vehicle` representing the truck route:

```
RESOURCE:
   identifier       :  Vehicle
   usage            :  sequential
   schedule domain  :  ScheduleHorizon
   activities       :  ClientVisit(c), StartAtDepot, EndAtDepot
   property         :  TransitionOnlyNext
   group set        :  Locations
   group definition :  ClientVisit(c)  : c,
                       StartAtDepot    : 'depot',
                       EndAtDepot      : 'depot'
   group transition :  (i,j) : DistanceInTime(i,j)
   first activity   :  StartAtDepot
   last activity    :  EndAtDepot
```

That is, the client visits cannot overlap in time, and moreover they have to respect the transition times (reflecting the distance between two visits).


```
empty AllVariables, Routes;
generateInitialRoutes;

repeat
  empty AllVariables;

  ! solve LP relaxation and get shadow prices
  solve FindOptimalRoutes where type := 'rmip';

  ! find new (improving) route within time limit
  solve FindNewRoute where time_limit := timeLimit;

  ! stop within 1 time unit (minute) of LP optimality
  break when ReducedCostOfRoute >= -1;

  ! otherwise, add new route to Master problem
  Routes += Card(Routes);
  for c | VisitClient.Present(c)  = 1 do
    ClientInRoute(c, Card(Routes)-1) := 1;
  endfor;
  LengthOfRoute(Card(Routes)-1) := EndAtDepot.End - StartAtDepot.Begin;

endrepeat;

! final solve for integer solution
solveAllocationIP;
```

NrClients = 30

TotalDistance = 1232

Selected Routes and Clients

	Length	c-01	c-02	c-03	c-04	c-05	c-06	c-07	c-08	c-09	c-10	c-11	c-12	c-13	c-14	c-15	c-16	c-17	c-
160	126					1													
410	73																		
473	168		1				1	1	1									1	
483	201			1	1											1			
486	190	1		1										1					
488	234							1	1		1	1			1		1		
494	151									1	1		1						
497	89				1														

a. Column generation procedure

b. Output page

Fig. 4. Column generation example for a vehicle routing problem.

To complete the model of our subproblem, we next define the truck capacity constraint and the objective:

```
CONSTRAINT:
   identifier :  CapacityConstraint
   definition :  sum(c, ClientVisit(c).Present*Load(c)) <= TruckCapacity

VARIABLE:
   identifier :  ReducedCostOfRoute
   range      :  free
   definition :  EndAtDepot.End - StartAtDepot.Begin
                 - sum(c, ClientVisit(c).Present*ClientServed(c).ShadowPrice)
```

Lastly, we define the column generation procedure as shown in Figure 4.a. After generating initial routes on single clients and pairs of clients in procedure `generateInitialRoutes`, we start the column generation process. We first solve the LP relaxation of the master problem `FindOptimalRoutes` (the keyword 'rmip' stands for relaxed MIP). We then solve the subproblem `FindNewRoute` with a given time limit. If the reduced cost is not improving, we exit the column generation loop. Otherwise, we add the newly found route to the set of routes. Here, `Routes` is an integer set starting with element 0, which allows to add the new route as element `Card(Routes)`, the cardinality of the set.

To illustrate again the visualization offered by Aimms, Figure 4.b shows the output page. It displays the depot and the clients in a Network object, together with the truck routes as arcs. The bottom table presents a detailed list of selected routes. We note that this visualization is dynamic, in that the displayed routes are updated each time a new solution is found (to this end, we solve `FindOptimalRoutes` as an additional MIP in the column generation procedure).

A detailed description of this application, together with the AIMMS model, can be downloaded from
`http://www.andrew.cmu.edu/user/vanhoeve/summerschool/exercises/`.

4.3 Inventory Balancing in Bike Sharing Systems

Our last example is a real-world application for inventory rebalancing and vehicle routing in in the context of bike sharing systems. Bike sharing systems have been installed in many major cities around the world. In these systems, users can pickup and return bikes at designated bike sharing stations with a finite number of docks. Unfortunately, user behavior results in spatial imbalance of the bike inventory over time. The system equilibrium is often characterized by unacceptably low availability of bikes or open docks, for pickups or returns respectively. Therefore, operators deploy a fleet of trucks to rebalance the bike inventory.

This problem consists of two main components. First, determining the desired inventory level at each bike station, which is typically done by an analysis of historic user data. Second, designing truck routes that will perform the necessary pickups and deliveries in order to reach the target inventory levels. In this example, we will assume for simplicity that we are given minimum and maximum target inventory thresholds for each bike station. The problem is then to find truck routes that will pickup and deliver the bikes at the visited stations such that the inventory level for each station is between its minimum and maximum threshold, with minimum total distance.

We can represent this problem as a CP scheduling problem with alternative resources; for each pair (i, j) of station i and vehicle j we define an optional activity representing whether j visits i. We use the global constraint `cp::Alternative` to ensure that at most one truck (or more) will perform the visit. Similar to the previous routing example we define a sequential resource for each vehicle representing its route. In addition, we need parallel resources for the bike stations and the vehicles to represent the inventory level over time. A complete description of the CP model can be found in [19].

One of the main benefits of the CP model for this application was the relatively fast development process. As described in [19], the CP model was compared with an exact MIP model as well as a heuristic MIP-based clustering approach. Even though the clustering approach outperforms the CP model in terms of solution quality (and solving time), it took several months to develop. In contrast, the CP model was implemented and tested within a few days, delivering competitive results in many cases.

Using AIMMS for this particular application demonstrated several advantages:

– We were able to develop all models (MIP, clustering heuristic, and CP) within one system. This greatly reduced the overhead of sharing data, as well as model development time.

– We were able to experiment with different MIP solvers for the same model, including CPLEX and Gurobi.
– Our application relied on large amounts of historic data that were processed on a separate server. We were able to quickly import necessary data using the external database functionality of Aimms.

5 Conclusions

We have presented the CP interface of the modeling system Aimms, and provided three examples that illustrate some of the benefits of developing CP applications using Aimms. The benefits include visualization of the results, representation of scheduling problems using activities and resources, and easy development of hybrid approaches such as CP-based column generation.

Acknowledgments

I would like to thank Chris Kuip and Marcel Hunting (Paragon Decision Technology) who developed the Aimms constraint programming interface, and provided support for earlier versions of this document. Also, the bike sharing application was developed together with Jasper Schuijbroek and Robert Hampshire.

Bibliography

[1] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. Kluwer Academic Publishers, 2001.
[2] J. Bisschop and A. Meeraus. On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study*, pages 1–29, 1982.
[3] Y. Colombani and S. Heipcke. Mosel: An Extensible Environment for Modeling and Programming Solutions. In *Proceedings of the Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR)*, 2002.
[4] R. Fourer and D.M. Gay. Extending an algebraic modeling language to support constraint programming. *INFORMS Journal on Computing*, 14: 322–344, 2002.
[5] R. Fourer, D.M. Gay, and B.W. Kernighan. AMPL: A Modeling Language for Mathematical Programming. *Management Science*, 36:519–554, 1990.
[6] A. M. Frisch, W. Harvey, C. Jefferson, B. M. Hernndez, and I. Miguel. Essence : A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.
[7] J.N. Hooker. *Integrated Methods for Optimization*. International Series in Operations Research & Management Science. Springer, 2007.

[8] U. Junker, S.E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A Framework for Constraint Programming Based Column Generation. In *Proceedings of CP*, volume 1713 of *LNCS*, pages 261–274. Springer, 1999.

[9] C. Kuip. The modeling system AIMMS for developing Mathematical Programming and Constraint Programming applications. In *CP Solvers: Modeling, Applications, Integration, and Standardization – CP2013 Workshop*, 2013.

[10] P. Laborie. IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems. In *Proceedings of CPAIOR*, volume 5547 of *LNCS*, pages 148–162. Springer, 2009.

[11] K. Marriott, N. Nethercote, R. Rafeh, P.J. Stuckey, M. Garcia De La Banda, and M. Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[12] L. D. Michel. Constraint Programming and a Usability Quest. In *Proceedings of CP*, volume 7514 of *LNCS*, page 1. Springer, 2012.

[13] M. Milano, editor. *Constraint and Integer Programming - Toward a Unified Methodology*, volume 27 of *Operations Research/Computer Science Interfaces Series*. Kluwer Academic Publishers, 2004.

[14] M. Milano and P. Van Hentenryck, editors. *Hybrid Optimization – The Ten Years of CPAIOR*, volume 45 of *Springer Optimization and Its Applications*. Springer, 2011.

[15] J.-F. Puget. Constraint Programming Next Challenge: Simplicity of Use. In *Proceedings of CP*, volume 3258 of *LNCS*, pages 5–8. Springer, 2004.

[16] M. Roelofs and J.J. Bisschop. *AIMMS 3.13: The Language Reference*. Paragon Decision Technology, 2012.

[17] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier, 2006.

[18] L.-M. Rousseau, M. Gendreau, and G. Pesant. Solving small VRPTWs with Constraint Programming Based Column Generation. In *Proceedings of CPAIOR'02*, 2002.

[19] J. Schuijbroek, R. Hampshire, and W.-J. van Hoeve. Inventory Rebalancing and Vehicle Routing in Bike Sharing Systems. Technical Report Tepper School of Business Working Paper 2013-E1, Carnegie Mellon University, 2013. Under Review.

[20] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999. with contributions by Irvin Lustig, Laurent Michel, and Jean-François Puget.

[21] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[22] T. Yunes, I.D. Aron, and J.N. Hooker. An integrated solver for optimization problems. *Operations Research*, 58(2):342–356, 2010.