High Level Rule Modeling Language for Airline Crew Pairing

Erdal Mutlu*, Ş. İlker Birbil†, Kerem Bülbül† and Hüsnü Yenigün*

*Sabancı University, Computer Science & Engineering Program, Orhanlı-Tuzla, 34956 Istanbul, Turkey
†Sabancı University, Manufacturing Systems & Industrial Eng. Program, Orhanlı-Tuzla, 34956 Istanbul, Turkey

Abstract. The crew pairing problem is an airline optimization problem where a set of least costly pairings (consecutive flights to be flown by a single crew) that covers every flight in a given flight network is sought. A pairing is defined by using a very complex set of feasibility rules imposed by international and national regulatory agencies, and also by the airline itself. The cost of a pairing is also defined by using complicated rules. When an optimization engine generates a sequence of flights from a given flight network, it has to check all these feasibility rules to ensure whether the sequence forms a valid pairing. Likewise, the engine needs to calculate the cost of the pairing by using certain rules. However, the rules used for checking the feasibility and calculating the costs are usually not static. Furthermore, the airline companies carry out what-if-type analyses through testing several alternate scenarios in each planning period. Therefore, embedding the implementation of feasibility checking and cost calculation rules into the source code of the optimization engine is not a practical approach. In this work, a high level language called ARUS is introduced for describing the feasibility and cost calculation rules. A compiler for ARUS is also implemented in this work to generate a dynamic link library to be used by crew pairing optimization engines.

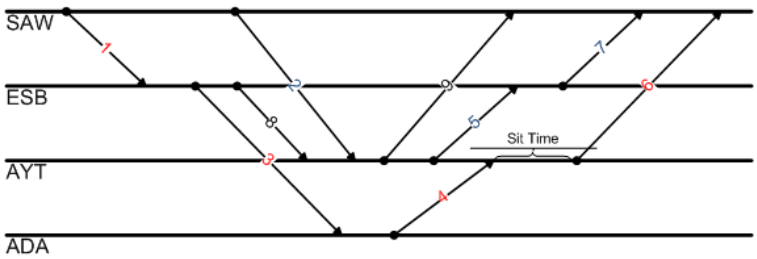
Keywords: Domain specific languages; Crew pairing; Rule Modeling
PACS:

INTRODUCTION

Crew costs are the second major cost item for airline companies after the fuel costs. This fact makes the optimization of crew costs very attractive for airline companies. Factors like layovers at non–base cities, transporting crew on flights as passengers (known as deadheading), and so on, have major impacts on crew costs. The airline crew scheduling problem is usually solved in two consecutive steps: crew pairing and rostering. In the crew pairing step anonymous crews are assigned to the flights, and in the rostering step the actual crew members are named for these anonymous teams. The major cost factors given above are mostly related to the crew pairing problem, since for many airlines, the crews are paid a fixed amount aside from the actual hours they fly.

The optimization problem solved in the crew pairing step is as follows. A *pairing* is a sequence of consecutive flight legs that starts from and ends at the base airport of the airline company, and it is flown by a single crew. Finding the least costly set of pairings that covers every flight in the given network is called the crew pairing problem. Figure 1 illustrates a simple flight network. Assuming that SAW is the base airport, we can have pairings like $p_1 = \langle 1,3,4,6\rangle$, $p_2 = \langle 2,5,7\rangle$, and $p_3 = \langle 1,8,9\rangle$. Note that these pairings cover all the flights in the given network. Since flight 1 exists both in pairing p_1 and pairing p_3, it will be flown by the crew of one of these pairings and the crew of the other pairing will then be deadheading.

Besides originating from and terminating at the base airport, there are numerous other constraints that a sequence of flights has to satisfy before it is considered as a pairing (see for example [1, 2] for a survey on such constraints). These feasibility rules are imposed by international and national regulatory agencies like IATA and JAR. In addition, airline companies have their own set of rules that they like to



1: A small flight network

use. A pairing is typically viewed as a sequence of duty periods (the sequence of flights flown by a crew in a working day). The rules are given, in general, to describe the feasibility of a duty (as to which sequence of flights forms a valid duty), the feasibility of a pairing (as to which sequence of duties forms a valid pairing) and the feasibility of a solution (as to which set of pairings forms a valid solution). In addition to the feasibility rules, there are also rules that explain how the cost of a pairing solution should be calculated. An optimization engine for crew pairing problem typically generates sequences of flights and then for each sequence generated, it checks the feasibility with respect to these rules. Likewise, the cost of a solution is also found by using the cost calculation rules.

Although the rules introduced by regulatory agencies are somewhat static, the ones that are applied by the airline companies are not. An airline may try several feasibility or cost scenarios by adding some other rules to find the least costly solution. There are different approaches on integrating these rules into a crew pairing optimization engine. One obvious way is to hard-code the rules into the engine. However, the modification of the rules then becomes quite difficult, since each scenario tried will also require some modification on the source code of the engine.

Another and a more flexible approach is to separate the description of the rules from the crew pairing engine. Jeppesen's CARMEN Crew Pairing System uses a special purpose language for expressing the rules that is known as Carmen Rave Language [3, 4]. With the use of a specific language, the end-users are entitled not only to change the rule data but also modify the structure of the rules without changing the crew pairing engine itself. Another system that uses a similar modeling language is DAYSY (Day-to-Day Resource Management Systems) rule handling system [5, 6]. This system defines a high-level object-oriented generic language DRL (DAYSY Rule Language) [7, 8] that can be used by different application domains for resource management systems.

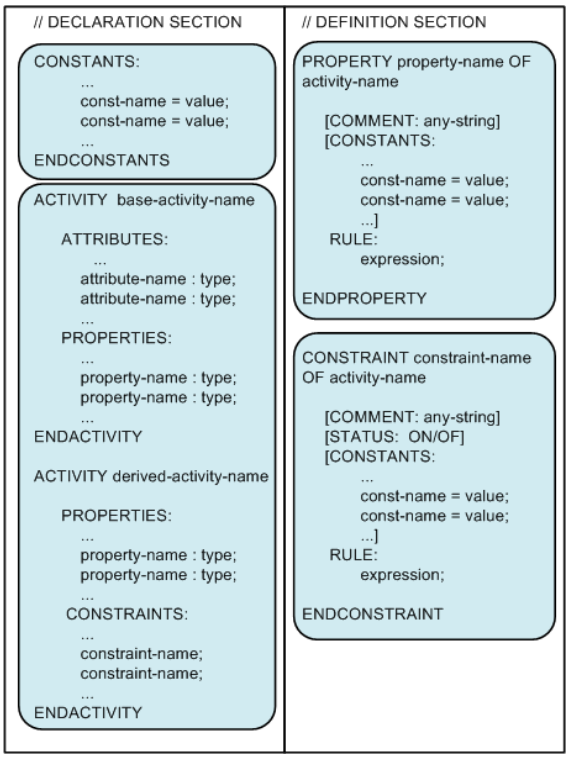
In this work, we introduce a high level language ARUS (Algopt RUle Specification language) [9] for modeling the feasibility/cost calculation rules for airline crew pairing problem. We use the general structure of the generic rule modeling language introduced in [8] and augment this language with new expressions and modifications according to the airline crew pairing problem. By using ARUS, users can define the hierarchical activity structure used in airline domain and can associate feasibility rules and cost calculation methods for each activity type.

In the following sections, the general structure and the main features of ARUS are described.

LANGUAGE DESIGN

In the crew pairing problem, we have a hierarchy of activities. At the lowest level as the *basic activity*, we have the flights. A *derived activity* is an activity composed of activities at a lower level. As the derived activities, we have duty (being composed of flights), pairing (being composed of duties), and solution (being composed of pairings). The feasibility rules specify the constraints that a set/sequence of activities at one level has to satisfy to form an activity of a higher level. As ARUS is designed for the airline crew pairing domain, activities from this domain such as flights, duties, pairings, and solutions are directly supported. In other words, these activities are built-in in ARUS and the properties, constraints and cost calculation rules regarding these activities can be given directly.

The general structure of a rule specification file, an ARUS program, is shown in Figure 2. In programming languages, it is a common practice to let the programmer declare all the entities, such as; variables, functions, and so on. Some properties of the declared entity are given at the declaration section. Take variable declaration, where the type of a variable is provided. Such a declaration provides a very simple control mechanism for possible programming errors. For instance, a variable declared to be of type integer can later be used only as an integer in the program; otherwise, it is an indication of a possible programming error. A type checker catches all such type mismatches. Following this declaration-use approach of general programming languages, ARUS also requires all entities to be declared and later used in the allowable contexts.



2: Specification file structure

An ARUS specification is composed of two parts: *declarations* and *definitions*. In the declarations part, the entities are introduced. The definitions part is where the descriptions of these entities are specified. In the next subsections, we give a brief explanation of these parts of ARUS specifications.

Data Types

The data types supported by ARUS can be classified as *general* and *domain specific*. The general data types are typical data types that are supported by general purpose programming languages. We have four general data types: `integer`, `real`, `string`, and `boolean`. The domain specific data types, on the other hand, are types that are specific to the airline crew pairing domain. There are four domain specific types which are `duration`, `time`, `datetime`, and `airport`. `duration`, `time` and `datetime` correspond to date-time types used for representing the time duration or a specific time and date. The airport type corresponds to 3 character IATA codes of the airports.

Declarations

The declaration part of an ARUS specification consists of two different types; *constant* and *activity* declarations. Each ARUS specification has one global constant declaration section, where some global constants are declared. Activity declarations part is used to give the declarations of basic and derived activities. As seen in Figure 2, an activity declaration has certain components that have to be declared: property, attribute and constraint declarations.

Property Declarations A property of an activity is a value to be computed for that activity. For example, the number of flights in a duty activity is a property of that duty activity. For each property defined, the type of the property must be given. A special property named `cost` must exist for each activity. This property is used to define the cost of the activity.

Attribute Declarations Some properties of *basic* activities are not computed but simply taken from the input. For example, departure time of a flight is such a property. These properties are called *attributes* and they have to be declared as well. A derived activity on the other hand cannot have an attribute, meaning all properties of derived activities have to be computed. An attribute is declared exactly in the same way as a property. A basic activity can have multiple attributes.

Constraint Declarations There are certain constraints that a derived activity has to satisfy. An example constraint could be "at most one flight departs from the base airport in a duty." Each such constraint has to be declared. A derived activity can have multiple constraints and all of those constraints have to be satisfied. A constraint is in fact a boolean property but semantically treated in a special way. The basic activities cannot have constraints.

Definitions

The definitions of the declared properties and constraints are provided in the definitions part. In other words, the computation of the property or the constraint of an activity is explained. Since a constraint of an activity is actually a boolean property, both constraint and property definitions have similar building sections. Property/constraint definitions can include a comment section, where users can include comments about the property/constraint. Likewise each property/constraint can have a constant section for declaring local constants. A rule definition section gives the actual computation method of the property/constraint. The only difference between a property and a constraint definition is that a constraint definition has a status section which is used for enabling/disabling the corresponding constraint check.

We now give an example rule specification in ARUS. One of the typical feasibility conditions is to have a minimum sit time between any two consecutive flights in a duty. This can be specified in ARUS as given in Rule 1.

The constraint defines the calculation method for each consecutive flight pair `f1` and `f2` in the elements of a duty. It simply subtracts the `arrival_time` of the latter flight from the `departure_time` of the former flight and checks if the result is more than the `minSitTime` which is set to 30 minutes.

A compiler is also implemented for ARUS to translate an ARUS specification into a dynamic link library. For each derived activity, this library exports two functions named `isFeasible` and `cost`. The first function simply checks

Rule 1 Minimum sit time between two consecutive flights in a duty cannot be less than 30 minutes

```
CONSTRAINT minimumSitTime OF Duty
   STATUS: ON;
   CONSTANTS:
     minSitTime = 00:30;
   RULE:
     FOR EACH f1 -> f2 IN ELEMENTS
       departure_time OF f2 - arrival_time of f1 >= minSitTime
ENDCONSTRAINT
```

if all the constraints (with the status set to ON) are satisfied or not. The cost function returns the value computed by the cost property defined for that activity.

CONCLUSION

In this work, we have designed a high level domain specific language, ARUS, to be used for specifying the feasibility and cost calculations in airline crew pairing. The language has its syntactic features inherited from DRL [8]. ARUS specializes DRL in a domain specific manner. Using such a high level language simplifies the task of incorporating these rules into the pairing system, without a need for a modification on crew pairing optimization engine.

ARUS is a work-in-progress. However we believe that even with the current form of ARUS, one can model quite a wide range of feasibility and cost calculation rules. In order to test this assertion, we are currently implementing the entire rule set for crew pairing of a local airline company in ARUS. We also believe that the syntax of ARUS is more intuitive and the declarative nature of the language makes it easier to be understood and to be used (especially by those who don't have a programming background, like planning departments' staff of airline companies) than the other available languages for the same task. In order to test this assertion, we need to perform a usability test.

ACKNOWLEDGMENTS

This research has been supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant 106M472.

REFERENCES

1. C. Barnhart, A. Cohn, E. Johnson, D. Klabjan, G. Nemhauser, and P. Vance, "Airline Crew Scheduling," in *Handbook of Transportation Science*, edited by F. S. Hillier, and R. W. Hall, Springer US, 2003, pp. 517–560.
2. B. Gopalakrishnan, and E. Johnson, *Annals of Operations Research* **140**, 305–337 (2005).
3. C. A. Hjorring, S. E. Karisch, and N. Kohl, "Carmen Systems' Recent Advances in Crew Scheduling," in *Proceedings of the 39th Annual AGIFORS Symposium*, 1999, pp. 404–420.
4. C. A. Hjorring, and J. Hansen, "Column Generation with a Rule Modelling Language for Airline Crew Pairing," in *Proceedings of the 34th Annual Conference of the Operational Research Society of New Zealand*, 1999.
5. C. Goumopoulos, P. Alefragis, K. Thrampoulidis, and E. Housos, "A Generic Legality Checker and Attribute Evaluator for a Distributed Enterprise Environment," in *Proceedings of the third IEEE International Symposium on Computers and Communications*, 1998, pp. 286–292.
6. K. Thrampoulidis, C. Goumopoulos, and E. Housos, "Rule handling in the day-to-day resource management problem: an object-oriented approach," in *Proceedings of the 5th Panhellenic Conference on Informatics*, 1995, pp. 821–830.
7. C. Goumopoulos, and E. Housos, *Journal of Systems and Software* **69**, 43 – 56 (2004).
8. K. X. Thrampoulidis, N. Diamantopoulos, and E. Housos, *Softw., Pract. Exper.* **27**, 1135–1161 (1997).
9. E. Mutlu, *A High Level Rule Modeling Language for Airline Crew Pairing: Design and Implementation*, Master's thesis, Sabanci University (2011).