HIGH LEVEL RULE MODELLING LANGUAGE FOR AIRLINE CREW PAIRING: DESIGN AND IMPLEMENTATION

by Erdal Mutlu

Submitted to the Graduate School of Sabancı University
in partial fulfillment of the requirements for the degree of
Master of Science

Sabanci University

March, 2011

HIGH LEVEL RULE MODELLING LANGUAGE
FOR AIRLINE CREW PAIRING:
DESIGN AND IMPLEMENTATION

APPROVED BY:

Assist. Prof. Dr. Hüsnü Yenigün	.....
(Thesis Advisor)	
Assist. Prof. Dr. Kerem Bülbül	.....
(Thesis Co-advisor)	
Assoc. Prof. Dr. Ş. İlker Birbil	.....
Assoc. Prof. Dr. Erkay Savaş	.....
Assoc. Prof. Dr. A. Berrin Yanıkoğlu	.....

DATE OF APPROVAL: .....

HIGH LEVEL RULE MODELLING LANGUAGE FOR AIRLINE CREW PAIRING: DESIGN AND IMPLEMENTATION

Erdal Mutlu

CS, Master's Thesis, 2011

Thesis Supervisors: Hüsnü Yenigün, Kerem Bülbül

Keywords: Programming Languages, Compiler Design, Crew Pairing Problem

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 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 using some 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 understand if the sequence is a valid pairing, and has to calculate the cost of the pairing by using the cost calculation rules. However the feasibility and cost calculation rules are not usually stable. Airline companies try several scenarios in each planning period. In this work, a high level language for describing the feasibility and cost calculation rules is designed. Airline companies can use such a domain specific language to

specify the rules for feasibility and cost calculation. A compiler for this language is also implemented which generates a dynamic library implementing the specified rules.

HAVAYOLU EKİP EŞLEME PROBLEMİ İÇİN ÜST SEVİYE KURAL MODELLEME DİLİ: TASARIM VE UYGULAMA

Erdal Mutlu

CS, Yüksek Lisans Tezi, 2011

Tez Danışmanları: Hüsnü Yenigün, Kerem Bülbül

Anahtar Kelimeler: Programlama Dilleri,
Derleyici Tasarımı, Ekip Eşleme Problemi

Özet

Ekip eşleme problemi, uçuş ağındaki her bir uçuşu kapsayan en az maliyetli eşleme (tek bir ekip tarafından uçurulan ardışık uçuşlar) kümesinin arandığı bir havayolu optimizasyon problemidir. Bir eşleme, uluslararası ve ulusal kural koyucular ve havayolu şirketinin kendisi tarafından düzenlenen bir çok karmaşık geçerlilik kurallar kümesi kullanılarak tanımlanır. Bir eşlemenin maliyeti de bazı yine karmaşık kurallar kullanılarak tanımlanır. Ne var ki bu kurallar sabit değildir. Havayolu şirketleri her planlama döneminde çeşitli senaryolar denerler. Bu çalışmada, geçerlilik ve maliyet hesaplama kurallarının tanımlanmasında kullanılacak bir üst düzey dil tasarlanmıştır. Böyle bir alana özgü dil kullanılarak, havayolu şirketleri geçerlilik ve maliyet hesaplama kurallarını kolayca belirtebilir. Bu dil için, geçerlilik kontrolü ve maliyet hesaplama fonksiyonları sağlayan bir dinamik kütüphane oluşturan bir derleyici de gerçekleştirilmiştir.

Acknowledgements

I wish to express my gratitude to my supervisor Hüsnü Yenigün, for his invaluable guidance, support and patience all through my work. I am also grateful to Kerem Bülbül and İlker Birbil, for all their support and guidance.

I also like to thank Erkay Savaş, Albert Levi, Cemal Yılmaz, Güvenç Şahin and Dilek Tüzün Aksu. It was wondeful to work with such brilliant professors. And I want to thank my colleagues İbrahim Muter, Duygu Taş, Nimet Aksoy for their support on my work.

Special thanks goes to my friends Erman Pattuk, Oya Çitci, İbrahim Boylu, Gülčin Çoşkun, Ahmet Onur Durahim, Duygu Karaoğlan, Emine Dumlu, Leyli Javid, Barış Altop, Murat Ergun, Hüseyin Ergin, Oya Şimşek, Cengiz Örencik, İsmail Fatih Yıldırım, Osman Kiraz, Emre Kaplan, Yarkın Doröz, Zafer Özcan, Tuğçe Yazıcıgil, Alper Ergin and many many others. And of course I owe my most sincere thanks to my dear Burcu Özçelik for her patience and loving support.

The last and the most important thanks goes to my family, who gave me enough support through all my education. Mübeyyen, Yakup Mutlu and Mübeccel Şahin.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Software applications are mostly used by end-users who have no or little programming skills. As a result when there is a problem or a need for modifications on the application, it is addressed to a specialized programmer who tries to transform the requirements of the end-user to applications. This approach results in a waste of time and resources for even little changes. This waste is even more drastic in cases that several different configurations has to be tested with incremental modifications on the application. A good example for this type of scenarios is in crew pairing domain where the solution depends on feasibility constraints. These constraints can be determined by government, employee union or company itself and can change frequently. In this type of applications, it is crucial that these changes can be made by end-users. This can be done by having a high level language that can be used by the end-users without a requirement of having good programming skills. The use of a high level rule modeling language can enable the end-user to express and modify the rules that determines the feasibility and cost calculations in the application easily.

In this work, we designed a high level language ARUS (Algopt RUle Specification language) used for modeling the feasibility/cost calculation rules for airline crew pairing problem in the context of "Robust airline crew pairing; models and solution algorithms" project which is supported by "Scientific and Technological Research Institution of Turkey" (TÜBİTAK). ARUS is a domain specific language specializing on the description of feasibility/cost calculation rules for the airline crew pairing problem. We also implemented a compiler for ARUS which generates a dynamically linked library of methods

for feasibility checking and cost calculation rules given in an ARUS specification.

1.1 Crew Pairing Problem

The crew pairing problem is the process of creating a set of pairings (also called as trips) that cover every flight in a given flight network [1]. Figure 1 gives to a small flight network. The most typical feasibility rule for a pairing is that it has to start from and end at the base (a home airport/city for the airline). Assuming that SAW is the home airport in Figure 1, the flight sequence $p_1 = \langle 1, 3, 4, 6 \rangle$ is a such a sequence. If it also fulfills the other requirements for being a pairing than this sequence is called a pairing. Some of the other such sequences in Figure 1 are $p_2 = \langle 2, 9 \rangle$, $p_3 = \langle 2, 5, 7 \rangle$, $p_4 = \langle 1, 8, 6 \rangle$ and $p_5 = \langle 1, 8, 9 \rangle$. The set of pairings $\{p_1 = \langle 1, 3, 4, 6 \rangle, p_3 = \langle 2, 5, 7 \rangle, p_5 = \langle 1, 8, 9 \rangle\}$ covers all the flights in the flight network and therefore it is called a (pairing) *solution*. In this solution, flight 1 appears in more than one pairing. This means the crew of one of these pairings (say in p_1) will be operating the flight whereas the crew of the other pairings (p_5 in this example) will be deadheading on that flight. In other words, they will be traveling as a passenger. A pairing is composed of sequence of duties, where a *duty* is the sequence of flights flown in a working day. Between two duties the crew members rests, which has to be in a city that is not the home base by definition. For example, the pairing $\langle 1, 8, 6 \rangle$ might be consisting of two duties where flights 1 and 8 are in the first duty, and the flight 6 is in the second duty.

While the coverage requirement is the primary concern, legality con-

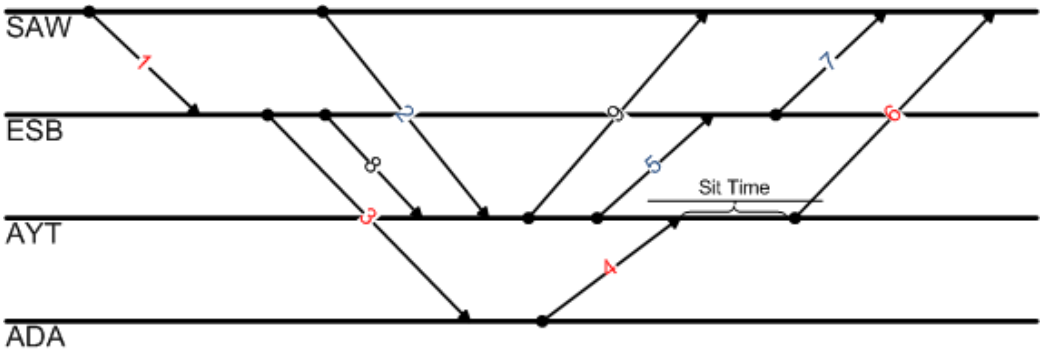

Figure 1: Flight Network

straints also have an important impact on the solution. In most of the crew pairing domains, such as rail or air travel, there are numerous rules or regulations. While some of these rules are fixed internationally, most of them tend to change from country to country or company to company according to their policies. These rules are mostly defined over the activity types of the domain. For example in airline crew pairing domain, the rules describe a feasible duty, a feasible pairing, and a feasible solution.

The complexity and dynamic changing nature of the rules and the regulations create the need for an efficient and user-friendly way to express and manage them. The main requirements for such a rule modeling system are [2];

- Correctness: Each rule modeled with the system has to be correct according to real life representation.

- Ease of modification: As the rules and regulations are frequently modified, it should be easy to change the rules which makes it possible to try what-if scenarios for different cases.

- Ease of development: The user should be able to use the system easily without a need for prior high skills.

There are different approaches to the problem, the traditional approach is based on "hard-coding" the rules into the crew pairing engine itself. Users, in this case, have to be experienced programmers, to be able to implement all the rules in a low level programming language. This approach makes it very hard for a user who do not have prior knowledge of the pairing engine or experience on a general purpose programming language to change or use the crew pairing engine. Also because of the correctness of a modeled rule is based on the skill of the implementor it is difficult to get it correctly implemented.

Another approach is based on using a rule modeling language, for representing the rules and regulations into the crew pairing engine. This approach makes it possible to change the rules frequently without doing changes on the crew pairing engine. There are some commercial examples that use this approach like Carmen from Jeppesen [2]. We will give details about these languages in Section 1.3.

1.2 Domain Specific Languages

A domain specific language (DSL) , is a language designed specifically for a domain that provides improvement in expressiveness and ease of use [3, 4]. Idea of designing new languages to be suited better in a domain is relatively new as general purpose languages have been the primary focus of language design [5]. Today, there are lots of examples for DSLs and Excel can be a good example since it is widely known and used. As the main property of

DSLs, Excel is particularly good for a specific number of problems, expressing plenty number of calculation methods to spreadsheet applications. Examples of DSLs, can be used in different domains, like AutoCAD for architectural design, Verilog for hardware description etc.

The key characteristics of DSLs are defined to be [5];

- The domain is well-defined and central.
- The notation is clear.
- The informal meaning is clear.
- The formal meaning is clear and is implemented.

In our case, we designed a high-level language called ARUS, for airline crew pairing problem to describe the rules for feasibility and cost calculation of pairings. With ARUS, we aim to make it easier for end-users who do not have any programming background, to interact with the system without doing changes on the engine. As it is crucial for a domain-specific language to be easy to understand by its users, in our design we preferred to use keywords from the terminology of the airline crew pairing domain.

1.3 Related Works

There are different approaches for integrating the legality rules and regulations into a tool generating these pairings. One way is to use "hard-coded" rules in the source code of the pairing generators. This makes the modification of the rules by the end-users (airline planning department) difficult. Since these rules and regulations can vary from company to company, or more

importantly, from time to time for the same airline company, it is important for the end-users to be able to describe these rules without having to dive into the source code of the pairing generators.

There are also systems like Jeppessen's CARMEN Crew Pairing System that uses a special purpose language, Carmen Rave Language [2, 6], for expressing the rules and constraints. With the use of a specific language, end-users can not only change the rule data but also modify the structure of the rules without changing the pairing generator tool itself. Another system that uses the help of a modeling language is DAYSY (Day-to-Day Resource Management Systems) rule handling system [7, 8]. They define a high-level Object-Oriented language called DRL (DAYSY Rule Language) [9, 10] that can be used by different application domains as it is a generic language designed for resource management systems.

1.4 Contribution of the Thesis

In this work, we designed a domain specific language to be used for the description of feasibility and cost calculation rules for airline crew pairing problem. We used the general structure of a generic rule modeling language introduced in [10] and added new expressions and modifications according to the airline crew pairing domain. Using this language, planning departments of airline companies can easily describe the feasibility and cost calculation rules without having to deal with the modifications on the source code of the pairing generators.

We also implemented a compiler for this language that can translate the rules given in this language into an equivalent C++ library. This library

is to be dynamically linked by the pairing generators. The library exports methods for checking the feasibility and calculating the cost of pairings.

1.5 Organization of the Thesis

In Section 2, we introduce our language ARUS in detail with its syntactic and semantic details. In Section 3, the implementation details of the ARUS compiler is given. Finally in Section 4, we give a conclusion together with some further improvements that can be made as future work. We also give some real life examples of feasibility rules for airline crew pairing problem with their corresponding representations in ARUS in Appendix A.

2 Language Design

A common feature in scheduling problems is the hierarchical structure of the activities. At the lowest level, there are *basic activities*. Activities at one level are combined to form an activity of a higher level. Activities that are formed by combining lower level activities are called *derived activities*. The feasibility rules give the constraints that a set/sequence of activities at one level has to satisfy to form an activity of a higher level. For the airline crew pairing problem, basic activities are flights, which are also called legs. Several flights are combined to form a duty, which is also called a shift. Similarly a sequence of duties are combined to form a pairing, or as it is sometimes called a trip. Finally a set of pairings form a solution.

A rule modeling language for a particular domain should support the activity hierarchy of that domain. As ARUS is designed for airline crew pairing domain, activities from this domain such as flights, duties, pairings, and solutions are directly supported. The language inherently assumes that flights combined to form a duty, duties are combined to form a pairing, and pairings are combined to form a solution.

Although activities at one level I are combined to form an activity of a higher level I', not every combination of activities of level I will be a valid/feasible activity for level I'. Certain constraints are enforced for a combination. For example the total time of a duty (which is a sequence of flights) cannot be longer than a certain limit. Therefore the rule language should let the users to define such constraints easily. The feasibility rules are based on some properties of the activities. Using the example above, the total time of a duty is a property of duty. The feasibility rule simply states

that it should be smaller than a given limit value. The "total time" of a duty is a property of the duty that needs to be calculated. The language should also make it easy for the user to state how such properties of activities are calculated.

ARUS allows description of properties of activities of airline crew pairing domain. Based on these properties, the constraints are stated easily. For each derived activity (duty, pairing, solution) one has to give the properties and the constraints of that activity. For the basic activity (flights), only properties have to be given. The description of a property or constraint of an activity is a function of the properties of the same and some other activities.

We now explain the general structure of a rule specification file, which is a program written in our rule language ARUS. In programming languages, it is common to let the programmer declare all the entities (e.g. variables, functions, etc.) used in a program. Some properties of the entity being declared are given at the declaration. For example in a variable declaration the type of a variable is provided. Such a declaration provides a simple control mechanism for possible programming errors. For instance, a variable declared to be of type integer can later be used in the program in the contexts where an integer value is required. Otherwise it is an indication of a possible programming error. A type checker aims to catch all such type mismatches to identify errors in a program. Following this declaration/use approach in general programming languages, ARUS also requires all entities to be declared and later used in allowable contexts. We give the general structure of specification file in Figure 2, which also shows the sections in this document

explaining different parts in an ARUS file.

A specification is composed of two parts: *declarations* and *definitions*. In the declarations part, a declaration of the entities are given. The definitions part is where the descriptions of these entities are specified. We will now give the details of ARUS. We start by introducing the data types supported by ARUS in Section 2.1. Declarations and definitions of ARUS are explained in Section 2.2 and Section 2.3, respectively. Finally in Section 2.4 ARUS expressions are given in a detailed manner.

2.1 Data Types

The data types supported by ARUS can be classified as *general data types* and *domain specific data types*. 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`, `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`. We now explain these data types in detail.

`integer`**:** This type is used to represent integers and it is not different than integer types available in general purpose programming languages. An integer literal consists of a sequence of digits with an optional minus sign for negative integers (e.g. `13`, `111`, `-12`, etc.). However leading zeros are not allowed, i.e. `0013` and `-023` are not valid ARUS integers. ARUS itself does not introduce any limit on the range of the integer values. However, an ARUS specification is converted into a C++ program. The limitations of the C++ compiler used to compile

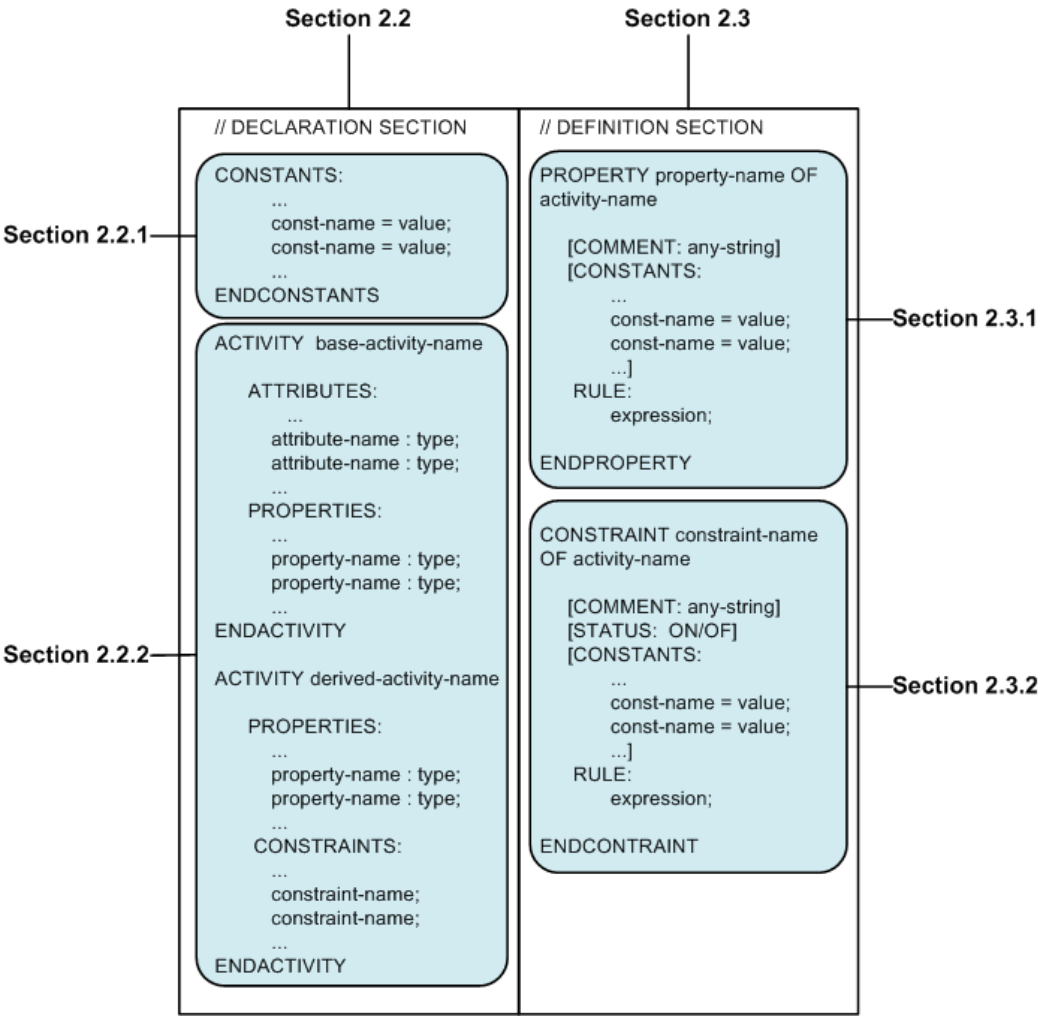

Figure 2: Specification File Structure

the generated C++ code may bring a range limitation on the integer values.

`real`: This type is used to represent real values. A real literal consists of a sequence of digits, followed by a dot, followed by a sequence of digits with an optional minus sign for negative numbers (e.g. `3.14`, `-123.2`, etc.). However integer part of a real number can not start with a zero except when the integer part is zero, and the fractional part can not end with a zero except when the fractional part is zero. For example, `0.12` and `12.0` are valid real numbers but `01.23` and `1.20` are not valid real numbers. ARUS itself does not introduce any limit on the range of the real values. However, an ARUS specification is converted into a C++ program. The limitations of the C++ compiler used to compile the generated C++ code may bring a range limitation on the real values.

`string`: This type is used to represent strings. A string is a sequence of characters (except the newline character) between quotation marks (e.g. `"a string"`, `"another string"`, etc.). The quotation mark itself is represented inside a string by escaping it using a backslash character (e.g. `"string with a \"quoted string\" can be defined"`).

`boolean`: This data type is used to represent the truth values of boolean logic. It only has two possible literal values: `true` and `false`.

`duration`: This data type is a domain-specific type. It is commonly used in the rules and regulations. It represents a basic time duration. The

general format for a duration representation is `H:M` where `H` is a non-negative integer representing the hour part and `M` is a two digit value in the range 00-59 representing the minutes part. For example `3:45` represents a duration of 3 hours and 45 minutes. For a duration less than hour, the hour part given as zero (e.g. `0:45`) and for a duration of an exact hour the minutes part has to be given as 00 (e.g. `13:00`).

`time`**:** This is another domain specific data type of ARUS. It is used to represent a specific time during a day but not on a specific date. For example, a feasibility rule may state that duties that start before 05:00AM in the morning should have a certain property. Constraints based on such time parameters are quite common in crew pairing problems. The general format of a `time` literal is `HH:MM` followed by either `AM` or `PM`. `HH` part is a two digit hour part in the range 00-12, `MM` part is a two digit minutes part in the range 00-59. Some valid `time` literal examples are `01:45AM`, `11:30PM`, `00:00AM` (midnight), `12:00PM` (noon). White space characters are allowed between `MM` part and `AM`/`PM`. Therefore, `12:45 PM` is also a valid `time` literal. Some invalid examples are `00:00PM` and `12:00AM` (must use `00:00AM` and `12:00PM` to denote midnight and noontime respectively), `11:75PM` (minutes part cannot be greater than 59), `14:00PM` (hour part cannot be greater than 12 and `02:00PM` must be used to denote two in the afternoon).

Note that, ARUS prefers to use 12-hour convention with AM and PM designators, rather than using 24-hour convention. The design decision behind this selection is to be able to uniquely and easily infer the type of literals. For example if ARUS had used 24-hour convention, then

`16:30` could be interpreted both as a `duration` value (16 hours and 30 minutes long time duration), and as a `time` value (half past four in the afternoon).

`datetime`: This type is used to represent a specific time on a specific date. The general format of a `datetime` literal is `dd.mm.yyyy` followed by `HH:MM` followed by either `AM` or by `PM`. Here `dd` is a two digit day specifier in the range 01-31, `mm` is a two digit month specifier in the range 01-12, and `yyyy` is a four digit year specifier. `HH:MM` is the same as explained in `time` type above. Some valid `datetime` literal examples are `03.02.2011 01:30 PM`, `29.02.2012 12:50PM`, `01.01.2011 00:00AM`. Some invalid examples are `32.12.2010 10:00AM` (day part cannot be greater than 31), `31.13.2010 10:00AM` (month part cannot be greater than 12). The same restrictions and conventions as in `time` type apply for `HH:MM AM/PM` part.

`airport`: This is another domain-specific data type for ARUS. It is used to represent airports. The literals of this data type are 3 character codes of the airports as defined by "International Air Transport Association"(IATA), also known in general as IATA codes of the airports. Example literals of this type are `IST` (Istanbul Atatürk International Airport), `SAW` (Sabiha Gökçen International Airport), `ATL` (Hartsfield-Jackson Atlanta International Airport), `AJI` (Ağrı Airport).

Besides the built-in data types introduced above, ARUS supports some type constructors. These type constructors are also similar to their counterparts in general purpose programming languages. However, their use is very

restricted in terms of the values that can be defined and also in terms of the operations allowed on such derived types compared to the general purpose programming languages.

Sets: This type constructor is similar to set type constructors available in general purpose programming languages to define a set of some other type. For example, one can define a set of integers, a set of strings, etc.

ARUS allows a very restricted form of set data type constructor. It is only possible to define a set type with a single constant value. In other words, only constant sets can be defined. The elements of a set value must all be of the same type. A set literal is given as a comma separated list of elements surrounded by { and }. For example, { `IST, SAW` } represents the set of airports in Istanbul. The operations on a set are also very restricted. Membership test operations ("belongs to" and "does not belong to") are allowed.

One can use this type constructor to create groups of the literals of the same type and use them in special expressions which will be explained in Section 2.4. There is no need for declaring the type of a set value. The type of a set value is easily inferred from the types of the elements which are all of the same type. Some valid set values are { `10:00 AM, 11:00PM, 02:35 AM` } (a set of `time` values), { `IST, SAW` } (a set of airports), { `"320", "321", "32M", "32L"`} (a set of strings representing aircraft types belonging to A320 fleet). Some invalid examples are {`00:40, 02:05AM` } and { `SAW, 14:45, IST` } which are both invalid due to mixed type of the elements.

Sequences: A sequence is a set where the order of the elements is important. Although ARUS has no concrete syntax for representing sequences, it is possible to generate sequence values. In this text, a sequence value is denoted by giving the arrow (`->`) separated elements of the sequence surrounded by { and }. For example { `SAW -> ADB -> AYT -> SAW` } is a sequence of airports.

Tables This data type constructor is similar to array type constructor in general purpose programming languages. ARUS has again a very restricted support due to the limited needs of the domain of airline crew pairing. If we use general array terminology, ARUS allows only one or two dimensional, constant arrays.

On the other hand, ARUS is more relaxed in terms of the conditions for matching indices given in array reference to a row and a column. In general purpose programming languages, an array reference of the form `A[x,y]` refers to row `i` and column `j` of the array iff `x==i` and `y==j`. Although implicit, each row/column has a predicate for being matched by an array reference but all these predicates are simple equality checks of the form given above. In ARUS, the predicate of a column/row is given explicitly as a boolean expression. For example, a column/row may have the following predicate for being selected: `(06:00AM <= dutyStartTime)`, where `dutyStartTime` is a value to be provided. A column/row is matched if its predicate evaluates to true by using the variable values used in the table reference (just like in an array reference a row/column `i` is matched when it's predicate `x==i` evaluates to true for `x=i`).

There may be multiple rows (columns, respectively) whose predicates evaluating to true for a table reference. In this case the first row (column, resp.) is considered to be the matching row (column, resp.). A special predicate `OTHERWISE` can be used as the predicate of the last row/column. Semantically this is equivalent to giving true as the predicate. Therefore, such a row/column is matched by default only when no other row/column has its predicate evaluating to true. If none of the predicates of rows/columns evaluate to true, then it's an invalid table reference (corresponding to an index out of bound case of array references in general purpose languages).

All the elements of a `TABLE` must be of the same type. The elements on a row are separated by "|" and rows are separated by a newline. In the first row and in the first column, the predicates of rows and columns are given, except the cell at the first row and the first column. In the cell at the first row and the first column, the list of variables used in the predicates must be given, by using `/` as a separator. This syntax is chosen due to established similar notations in the airline crew pairing domain.

An example `TABLE` declaration is given in Example 1 :

Here `maxDutyTime` is the name of the table, `numOfLegs` and `startTime` are the variables that will be used in the predicates of rows and columns. It is a 2×2 table. The first row has the predicate `1 <= numOfLegs <= 2` and the second row has the predicate `3 <= numOfLegs <= 5`. Similarly the predicate of the first column is `06:00AM <= startTime < 03:00PM`. The predicate of the last column is `OTHERWISE` hence it will

Example 1 Table Declaration example

```
TABLE maxDutyTime
numOfLegs/startTime| 06:00AM<startTime<03:00PM | OTHERWISE
1 <= numOfLegs <= 2|           14:00           |   12:00
3 <= numOfLegs <= 5|           13:00           |   10:00
```

only be matched if `startTime` is equal to or earlier than `05:59AM`, or later than `03:00PM` during the day. The elements of the array are of `duration` type. An example table reference can be given as `maxDutyTime[4,04:30PM]`. With `numOfLegs=4` and `startTime=04:30 PM`, we see that the last row and the last column are matched. Therefore this table reference will have the value `10:00`, i.e. a duration of 10 hours.

2.2 Declarations

As explained at the beginning of Section 2, an ARUS specification consists of two main parts: declarations and definitions. In this section, we explain declarations part in detail.

Constants and activities in an ARUS specification have to be declared before they are used. In the remainder of this section, keywords of the declaration part and the key features are detailed. We first introduce constant declarations in Section 2.2.1. We give details of activity declarations in Section 2.2.2.

2.2.1 Constant Declarations

Each ARUS specification has one global constants declaration section where (as the name implies) some global constants are declared. The section is marked by the keywords `CONSTANTS` and `ENDCONSTANTS`. Within the section, each constant is introduced by giving the name of the constant, followed by an equality sign, followed by the value of the constant, and the declaration is terminated by a semicolon. An example global constant declaration section is given in Example 2.

Example 2 An example global constants section

```
CONSTANTS
   briefingTime = 00:45;
   IstanbulAirports = {IST, SAW};
ENDCONSTANTS
```

The only exception in the syntax of a global constant declaration is in the declaration of a `TABLE` type constant. For these constants, the name is given within the declaration. Example 3 includes a case where a `TABLE` type constant is also declared.

As can be seen from this example, the name of the `TABLE` constant is `maxDutyTime` and it is given inside the `TABLE`.

2.2.2 Activity Declarations

There is a certain activity hierarchy in crew pairing problems. At the bottom of the hierarchy (as the basic activities) we have flights. Flight activities are

Example 3 An example for `TABLE` constant declaration

```
CONSTANTS
  briefingTime = 00:45;
  TABLE maxDutyTime
  numOfLegs/startTime |06:00AM<= startTime <03:00PM| OTHERWISE
  1 <= numOfLegs <= 2 |            14:00           |   12:00
  3 <= numOfLegs <= 5 |            13:00           |   10:00
  ;
  IstanbulAirports = {IST, SAW};
 ENDCONSTANTS
```

combined to form duty activities, duty activities are combined to form pairing activities. Finally pairings are combined to form solutions. Activities other than the basic activities (flights), are called *derived* activities.

Each activity has to be declared in an ARUS specification. An activity declaration starts with the keyword `ACTIVITY`, followed by the name of the activity (e.g. `FLIGHT`, `DUTY`, etc.). The declaration ends with the keyword `ENDACTIVITY`. An example activity declaration is given in Example 4, without providing any detail about the internals of an activity declaration.

Example 4 An Activity Declaration Example

```
 ACTIVITY Flight
   ...
 ENDACTIVITY
```

Inside an activity declaration there are certain components that have to be declared. These components are given below:

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. It is simply computed by counting the number of flights in the duty. Another example can be the time away from the base in a pairing, which can be computed as the arrival `datetime` of the last flight minus the departure `datetime` of the first flight in the pairing. Each property of an activity has to be declared inside the activity declaration. A property is declared by giving the name of the property followed by a colon, followed by the data type, and finally followed by a semicolon. An activity may have multiple properties.

Crew pairing problem is an optimization problem. The purpose is to minimize the cost of the solution. The cost of a solution is defined as the sum of the costs of the pairings in a solution. Therefore one has to explain how the cost of a pairing is defined. In ARUS, the cost of a pairing is defined by using a property named `cost`. Hence, the pairing activity must declare a property with this name. Other activities may or may not have such a property.

Attribute Declarations Some properties of basic activities are not computed but simply given in 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 activities.

Constraint Declarations As explained before, a derived activity I consists of a sequence/set of other activities I'. However, not every sequence of activities of type I' forms a derived activity I. To be more concrete, we say that a duty is a sequence of flights but not every sequence of flights will be considered as a duty. There are certain constraints that the sequence of flights has to satisfy to be a duty. An example constraint could be "at most one flight departs from the base airport in a duty". Another example could be "there cannot be more than 5 flights 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 (it is possible to deactivate some constraints but this will be explained in Section 2.3.2). A constraint is in fact a boolean property but semantically treated in a special way. A constraint declaration is performed by giving the name of the constraint followed by a semicolon. Note that basic activities cannot have constraints.

Inside an activity declaration, property/attribute/constraint declarations for that activity must be given. They are separated from each other by using the labels "`PROPERTIES:`", "`ATTRIBUTES:`", and "`CONSTRAINTS:`". Two complete activity declaration examples are given in Example 5 and Example 6.

Example 5 A Basic Activity Declaration Example

```
ACTIVITY Flight
  ATTRIBUTES:
    departureTime: datetime;
    arrivalTime: datetime;
    departureAirport: airport;
    arrivalAirport: airport;
    flightNo: string;

  PROPERTIES:
    flightTime: duration;
    isDomestic: boolean;
ENDACTIVITY
```

Example 6 A Derived Activity Declaration Example

```
ACTIVITY Duty
  PROPERTIES:
    totalDuration: duration;
    totalRestTime: duration;

  CONSTRAINTS:
    maxTotalDuration;
    maxTotalRestTime;
ENDACTIVITY
```

2.3 Definitions

In the definitions part, the definitions of properties and activities given in declarations part are provided. In other words, for a property/constraint of an activity, it is explained how this property/constraint is to be computed. Each property/constraint declared has to be defined.

2.3.1 Property Definition

In this section we will give details of a property definition. In ARUS a property definition is given by using the keyword `PROPERTY` followed by the name of the property, followed by the keyword `OF`, followed by the name of the activity to which the property belongs. The definition is terminated by the keyword `ENDPROPERTY`. In Example 7 a skeleton property definition is given without any details about the internals.

Example 7 A Simple Property Definition Example

```
PROPERTY overhead OF Duty
  ...
ENDPROPERTY
```

Inside a property definition, the following sections exist.

Comment Section One can give a free text explanation for the property inside the definition of the property. The keyword `COMMENT:` starts the section, followed by the free text explanation. The section is terminated by using a semicolon. This is an optional section.

Constant Declarations Section It is possible to declare some local constants to be used inside the property definition. The section starts with the keyword `CONSTANTS:` and terminated by the start of another section. Within this section any number of constant declarations can be given. A constant is declared by giving the name of the constant, an equality sign, an expression, and finally a semicolon. This is also an optional section.

Rule Definition Section In this section the expression that represents the calculation rule is given. The section starts with the keyword `RULE:` and terminated by the termination of the property definition (i.e. by the keyword `ENDPROPERTY`. Between the keyword `RULE:` and the keyword `ENDPROPERTY` an expression is given that explains how the value of the property is to be calculated. The type of the expression given in this part must match the type of the property given in its declaration. We will give detailed information about ARUS expressions in Section 2.4. Every property definition must have a rule section.

A complete property definition is given in Example 8. In this example, the expression "`briefingTime + debriefingTime`" gives how the value of the property "`overhead`" of a "`Duty`" activity is to be computed. In this example, the expression is simply based on constant values that are also defined in the property definition. However, in general this expression can be quite complex, referring to the other properties of the same duty and also to the properties of the flights in the duty. More advanced examples will be given later after introducing ARUS expressions.

Example 8 A Complete Property Definition Example

```
PROPERTY overhead OF Duty
  COMMENT: This is the overhead time that has
           to be added to every duty;
  CONSTANTS:
    briefingTime = 00:45;
    debriefingTime = 00:30;
  RULE:
    briefingTime + debriefingTime;
ENDPROPERTY
```

2.3.2 Constraint Definition

In this section we will give details of a constraint definition. In ARUS a constraint definition is given by using the keyword `CONSTRAINT` followed by the name of the constraint, followed by the keyword `OF`, followed by the name of the activity to which the constraint belongs. The definition is terminated by the keyword `ENDCONSTRAINT`. In Example 9 a skeleton constraint definition is given without any details about the internals.

Example 9 A Simple Constraint Definition Example

```
CONSTRAINT noEarlyDuty OF Duty
  ...
ENDCONSTRAINT
```

Inside a property definition, the following sections exist.

Comment Section One can give a free text explanation for the constraint inside the definition of the constraint. The keyword `COMMENT:` starts the section, followed by the free text explanation. The section is terminated by using a semicolon. This is an optional section.

Status Section Previously it was stated that all the constraints of a derived activity have to be satisfied. However, a constraint can be activated/deactivated easily by setting the status of the constraint to `ON` or `OFF`. The status section in a constraint definition is used for this purpose. The section starts with the keyword `STATUS:` followed by a status indicator (either `ON` or `OFF`) and terminated by a semicolon. This section is optional and when missing the constraint is assumed to be active.

Constant Declarations Section It is possible to declare some local constants to be used inside the constraint definition. The section starts by the keyword `CONSTANTS:` and terminated by the start of another section. Within this section any number of constant declarations can be given. A constant is declared by giving the name of the constant, an equality sign, an expression, and finally a semicolon. This section is optional.

Rule Definition Section In this section the expression that represents the calculation rule is given. The section starts with the keyword `RULE:` and terminated by the termination of the constraint definition (i.e. by the keyword `ENDCONSTRAINT`. Between the keyword `RULE:` and the keyword `ENDCONSTRAINT` a *boolean* expression is given that explains

how the value of the constraint is to be calculated. The constraint is satisfied iff the boolean expression evaluates to true for that duty. Every constraint definition must have a rule section.

A complete constraint definition is given in Example 10. In this example, a constraint of a duty activity is given. The constraint is used to make sure that no duty starts before 05:00AM in the morning. The example assumes that the duty activity has a property named `dutyStartTime`.

Example 10 A Complete Constraint Definition Example

```
CONSTRAINT noEarlyDuty OF Duty
  COMMENT: Duties must not start before 05:00AM;
  STATUS: ON;
  CONSTANTS:
    earliestDutyStartTime = 05:00AM;
  RULE:
    dutyStartTime >= earliestDutyStartTime;
ENDCONSTRAINT
```

2.4 Expressions

Inside the `RULE` section of a property/constraint definition, one has to give an expression that defines how the value of the property/constraint is computed. In this section, we explain ARUS expressions that can be used within property/constraint definitions.

Atomic Expressions By atomic expressions we mean expressions consisting of a single item, as opposed to complex expressions that are composed of sub-expressions combined by using operators.

Each literal is an atomic expression. For example, `integer` literal `7`, `string` literal `"LH"`, `airport` literal `SAW`, etc. are all atomic expressions. A constant (global or local) is also an atomic expression. Finally, a property or an attribute of an activity is an atomic expression. For example, suppose that `duty` activity has a property named `overhead`. The use of `overhead` itself is an atomic expression.

Atomic expressions are combined by using operators that will be defined below to form complex expressions.

Arithmetic Expressions The four common arithmetic operators addition (`+`), subtraction (`-`), multiplication (`*`) and division (`/`) are supported in ARUS for certain data types. Table 1 gives the contexts in which these operators are allowed and it also gives the type of the overall expression.

For example, one may want to define a property named `flight_durati on` for a flight, which is simply the duration of the flight. Such a property definition is given in Example 11.

In Example 11, `arrival_time` and `departure_time` are two attributes, and the value of the `flight_duration` property is defined to be simply `arrival_time - departure_time` which evaluates to a `duration` value.

Boolean Expressions ARUS supports logical and (`AND`), logical or (`OR`)

Type 1	Operator	Type 2	Result Type
integer	+,-,*	integer	integer
integer	/	integer	real
integer	+,-,*, /	real	real
real	+,-,*, /	integer	real
real	+,-,*, /	real	real
duration	+,-	duration	duration
duration	+,-	time	time
duration	+,-	datetime	datetime
time	+,-	duration	time
time	-	time	duration
datetime	+,-	duration	datetime
datetime	-	datetime	duration

Table 1: Arithmetic operators, allowed contexts, resulting types

and logical negation (NOT) operators on boolean values.

Relational Expressions ARUS supports the following relational operators: smaller than (<), smaller than or equal to (<=), greater than (>), greater than or equal to (>=), equal (==), and not equal (!=). Table 2 gives the contexts in which these operators are allowed. The type of the overall expression is boolean for all cases.

As a syntactic sugar, ARUS also supports chained relational expressions of the form "v_1 op_1 v_2 op_2 v_3 op_3 v_4 op_4 ..." as a shorthand notation for "(v_1 op_1 v_2) AND (v_2 op_2 v_3) AND (v_3 op_3 op_4)

Example 11 Setting `flight_duration` of a flight

```
PROPERTY flight_duration OF Flight
  RULE:
    arrival_time - departure_time;
ENDPROPERTY
```

`AND ...`". Here $\texttt{op}_i$'s are relational operators and $\texttt{v}_i$'s are values to be compared. As a more concrete example, `1 <= x < 5` is equivalent to `(1 <= x) AND (x < 5)`.

Set Membership The operators `IN` and `NOT IN` are used to test the membership of an element in a set. The syntax of expressions incorporating these operators are either "`e IN s`" or "`e NOT IN s`", where `e` is a value of a type (let's say `T`) and `s` is a set value defined over type `T`. A set membership expression itself has boolean type. "`e IN s`" evaluates to true only if the value `e` is one of the elements in the set `s`. "`e NOT IN s`" is equivalent to "`NOT (e IN s)`". A concrete example for a set membership expression is "`SAW IN { IST, SAW, AYT }`".

Conditional Expressions Conditional expressions in ARUS are given by using if-else expressions that are used in general purpose programming languages. The syntax of a conditional expression is

$$\texttt{IF (cond) } \texttt{e}_1\texttt{; ELSE } \texttt{e}_2\texttt{;}$$

Here `cond` is a boolean expression and $\texttt{e}_1$ and $\texttt{e}_2$ are two expressions that must be of the same type `T`. The semantics is exactly is the same

Type 1	Operator	Type 2
integer	<,<=,>,>=,==,!=	integer
integer	<,<=,>,>=,==,!=	real
real	<,<=,>,>=,==,!=	integer
real	<,<=,>,>=,==,!=	real
string	==,!=	string
boolean	==,!=	boolean
duration	<,<=,>,>=,==,!=	duration
time	<,<=,>,>=,==,!=	time
datetime	<,<=,>,>=,==,!=	datetime
airport	==,!=	airport

Table 2: Relational operators and allowed contexts

as the semantics in the other languages. If the condition `cond` evaluates to true, then the value of the conditional expression is the value of the expression $\texttt{e}_1$, otherwise it is the value of the expression $\texttt{e}_2$. The type of a conditional expression is the common type `T` of the expressions $\texttt{e}_1$ and $\texttt{e}_2$.

For example, suppose that a flight is to be classified as a domestic, a European, or a US flight. In order to be able to do this, one needs to check the flight's arrival/departure airport. Either the arrival or the departure airport is assumed to be domestic, which is correct for many airline companies. Suppose that we define sets of domestic, European and US airports as set constants with the names `DOMESTIC_AIRPORTS`,

EUROPEAN_AIRPORTS and US_AIPRORTS. Finally let arrival_airport and departure_airport be two attributes of a flight activity, which are read from the input flight data.

One can then define a property named flight_status by using the following expression:

Example 12 Setting flight_status of a flight

```
PROPERTY flight_status OF Flight
  RULE:
    IF (departure_airport IN EUROPEAN_AIRPORTS) OR
     (arrival_airport IN EUROPEAN_AIRPORTS)
   "EUROPEAN FLIGHT";
ELSE IF (departure_airport IN US_AIRPORTS) OR
         (arrival_airport IN US_AIRPORTS)
   "US FLIGHT";
ELSE
   "DOMESTIC FLIGHT";
ENDPROPERTY
```

Table References A table reference is given by the name of the table followed by the comma separated index values, surrounded by square brackets (e.g. maxDutyTime[4,04:30PM]). The semantics of table references are explained at the end of Section 2.1. The type of a table reference is the same as the common type of the elements of the table.

Operations on Sequences and Sets Since a derived activity is a sequen-

ce/set of other activities, the operations on sequences/sets of values are quite common in crew pairing domain. For example, the total flying time in a duty is calculated as the sum of the duration of the sequence of flights in that duty. To express such calculations, ARUS allows the following operators in sequences and sets. All the operators are unary and prefix operators.

COUNT OF: The `COUNT OF` operator takes a sequence/set as a parameter and returns the number of elements in that sequence/set. The value of a `COUNT OF` expression is `integer`. An example expression is given below where the result is integer value 3.

```
COUNT OF { 1:30, 2:20, 0:50 }
```

SUM OF: The `SUM OF` operator takes a sequence/set as a parameter and returns the sum of all the elements in that sequence/set. The elements of the sequence/set must all be of the same type `T`. It is also required that addition is defined between two elements of type `T`. The value of a `SUM OF` expression is also of type `T`. An example expression is given below where the result is a duration value `4:40`.

```
SUM OF { 1:30, 2:20, 0:50 }
```

AVG OF: The `AVG OF` operator takes a sequence/set `S` as a parameter and evaluates to the value `(SUM OF S)/(COUNT OF S)`, i.e. it evaluates to the average of the elements in `S`. The elements of the sequence/set must all be of the same type `T`. It is also required

that addition is defined between two elements of type `T`, and division of `T` over `integer` must be defined. The value of an `AVG OF` expression is of type `real`. An example expression is given below where the result is a real value `3.0`.

```
AVG OF { 1, 3, 5 }
```

`MAX OF:` The `MAX OF` operator takes a sequence/set `S` as a parameter and evaluates to the maximum value in `S`. The elements of the sequence/set must all be of the same type `T`, and they must be pairwise comparable. In other words, the operator `<=` must be defined between two values of `T`. The type of a `MAX OF` expression is `T`. An example `MAX OF` expression is given below. The value of the expression is the duration value `2:20`.

```
MAX OF { 1:30, 2:20, 0:50 }
```

`MIN OF:` The `MIN OF` operator takes a sequence/set `S` as a parameter and evaluates to the minimum value in `S`. The elements of the sequence/set must all be of the same type `T`, and they must be pairwise comparable. In other words, the operator `<=` must be defined between two values of `T`. The type of a `MIN OF` expression is `T`. An example `MIN OF` expression is given below. The value of the expression is the duration value `0:50`.

```
MIN OF { 1:30, 2:20, 0:50 }
```

`FIRST OF:` The `FIRST OF` operator takes a sequence (not a set) `S` as a parameter and evaluates to the first element in `S`. The elements of

the sequence must all be of the same type `T`. The type of a `FIRST OF` expression is also `T`. An example `FIRST OF` expression is given below. The value of the expression is the duration value `1:30`.

```
FIRST OF { 1:30 -> 2:20 -> 0:50 }
```

`LAST OF:` The `LAST OF` operator takes a sequence (not a set) `S` as a parameter and evaluates to the first element in `S`. The elements of the sequence must all be of the same type `T`. The type of a `LAST OF` expression is also `T`. An example `LAST OF` expression is given below. The value of the expression is the duration value `0:50`.

```
LAST OF { 1:30 -> 2:20 -> 0:50 }
```

Elements inside derived activities ARUS has a special keyword to refer to the set or the sequence of elements inside an activity. Please recall that a derived activity is a sequence or a set of other activities. A solution is defined as a set of pairings. A pairing is a sequence of duties, and a duty is a sequence of flights. The keyword `ELEMENTS` (depending on the context in which it is used) refers to the set or the sequence of elements inside in activity. If `ELEMENTS` is used inside a solution, it refers to the set of pairings in that solution. If `ELEMENTS` is used inside a pairing, it refers to the sequence of duties of that pairing. Finally if `ELEMENTS` is used inside a duty, it refers to the sequence of flights of that duty. `ELEMENTS` cannot be used inside flight activity.

For example, suppose that a duty cannot have more than 6 flights. This constraint can be stated as given in Example 13.

Example 13 A duty cannot have more than 6 flights

```
CONSTRAINT noLongDuties OF Duty
  RULE:
    COUNT OF ELEMENTS <= 6
ENDCONSTRAINT
```

Applying Properties to Activities For an instance of activity a and a property p of that activity type, "p `OF` a" gives the value of the property p of the instance a. For example, suppose that we want to calculate the total time of a duty. This can simply be computed the departure time of the first flight in the duty and the arrival time of the last flight in the duty. Assume that `departure_time` and `arrival_time` are properties (attributes in fact) of flights. We can then define the required property for a duty as shown in Example 14.

Example 14 Calculating total duty time

```
PROPERTY totalTime OF Duty
  RULE:
   (arrival_time OF LAST OF ELEMENTS) -
       (departure_time OF FIRST OF ELEMENTS)
ENDPROPERTY
```

In Example 14, `LAST OF ELEMENTS` evaluates to a flight. `arrival_time` property is applied on this flight, and this application evaluates to the value of the `arrival_time` property of that flight.

Applying Properties to Sequences/Sets of Activities For a sequence /set of activities, one can apply a property of that activity to generate a sequence/set of the values of that property of the elements of the sequence/set of activities. More formally, if s is a sequence/set of the form $\{\ o_1\ \overset{\texttt{->}}{,}\ o_2\ \overset{\texttt{->}}{,}\ \cdots\ \overset{\texttt{->}}{,}\ o_k\ \}$ of activities of type I (where I is pairing, duty, or flight), and p is a property of I, then "p `OF` s" is a sequence s' of the form $\{\ o'_1\ \overset{\texttt{->}}{,}\ o'_2\ \overset{\texttt{->}}{,}\ \cdots\ \overset{\texttt{->}}{,}\ o'_k\ \}$ where o'_i is equal to the value of p `OF` o_i. If the type of the property p is `T`, then s' is a sequence/set of values of type `T`.

For example, suppose that `ELEMENTS` is used inside a duty, hence it refers to the sequence of flights of that duty. Suppose that we want to calculate total flytime of that duty which is simply the sum of durations of the flights in this sequence. Assuming that `flight_duration` property of a flight is defined as given in Example 11, one can define the total flytime of a duty as shown in Example 15.

Example 15 Total flytime of a duty

```
PROPERTY totalFlytime OF Duty
  RULE:
    SUM OF flight_duration OF ELEMENTS
ENDPROPERTY
```

In Example 15, `ELEMENTS` generates a flight sequence s. `flight_duration OF ELEMENTS` generates a sequence s' of duration values where the i^{th} element in s' is the duration of the i^{th} flight in s. Finally, `SUM OF` operator adds up the elements in s'.

`WHERE` expressions A subsequence/subset of elements of a sequence/set can be selected by using `WHERE` expressions. The syntax of a `WHERE` expression is

$$s \texttt{ WHERE } exp\texttt{;}$$

where s is a sequence/set of objects (i.e. a sequence of flights or duties, or a set of pairings) and exp is a boolean expression. The boolean expression may refer to the properties of the objects in s. The semantics of a `WHERE` expression is as follows. Suppose s is a sequence/set of objects $\{o_1 \overset{->}{,} o_2 \overset{->}{,} \ldots \overset{->}{,} o_k\}$. Then s `WHERE` exp is a subsequence/subset s' of s, where only those elements in s for which the boolean expression exp hold survive in s'. If s is a sequence, then the relative order of the elements are preserved in s'. While checking if exp evaluates to true for an object o_i, if exp refers to a property p, then p `OF` o_i is understood.

For example, `ELEMENTS` used in a duty refers to the entire flight sequence of the duty. Using `ELEMENTS` together with `WHERE`, one can generate a sequence of flights of the duty where only international flights are considered. In order to illustrate a use of this construct, suppose we want to state a constraint which says "there cannot be more than one international flight" in a duty. This constraint can be given as shown in Example 16 by assuming the existence of `flight_status` property of flights as given in Example 12.

Here, `ELEMENTS` generates the flight sequence in a duty. This sequence is operated on by `WHERE` whose boolean expression is `(flight_status != "DOMESTIC_FLIGHT")`. Therefore, each flight in the sequence `ELEMENTS`

Example 16 A duty cannot have multiple international flights

```
CONSTRAINT noMultipleInternationalFlights OF Duty
  RULE:
    COUNT OF ELEMENTS
             WHERE (flight_status != "DOMESTIC_FLIGHT") <= 1
ENDCONSTRAINT
```

will be considered one by one to see if it satisfies this boolean condition. While checking the boolean condition for a flight f, the property `flight_status` will be understood as the value of `flight_status OF` f.

`FOR EACH` **operator** Another common type of constraints/properties in airline crew pairing domain concerns the relation between some consecutive activities inside a derived activity. For example, between two consecutive flights in a duty, there has to be a sufficient time lag called minimum sit time. In order to be able to state such constraints, ARUS provides `FOR EACH` iterator over a sequence of items. The syntax of this operator is

`FOR EACH` $sequence\_identifiers$ `IN` s exp

where exp is a boolean expression and s is a sequence. $sequence\_identifiers$ part denotes the number of consecutive elements to be considered in s and also give names to these elements that will be used to refer to them in exp. The syntax of $sequence\_identifiers$ is

e_1 -> e_2 -> ... -> e_k

where e_i is an identifier that will be used to refer to an element in *exp*. k is the number of the consecutive elements to be considered. For example, "f1 -> f2 -> f3" means that all three consecutive elements will be considered, and f1, f2, and f3 will be used as names while referring to the three consecutive elements.

The boolean expression *exp* incorporates e_i's as free variables. For each consecutive list of k elements, *exp* is evaluated separately. The value of the overall FOR EACH expression is the conjunction of the values of each of these boolean expressions.

As an example, suppose that between two consecutive flights in a duty there has to be a minimum of 60 minutes. This constraint can be stated as shown in Example 17.

Example 17 Minimum sit time between the flights in a duty

```
CONSTRAINT minimumSitTime OF Duty
  STATUS: ON;
  CONSTANTS:
    minSitTime = 1:00;
  RULE:
    FOR EACH f1 -> f2 IN ELEMENTS
      departure_time OF f2 - arrival_time of f1 >= minSitTime
ENDCONSTRAINT
```

As another example, suppose that for every consecutive three duties in

a pairing, there has to be at least one short duty, where a short duty is defined as a duty with a total flying time of less than 4 hours. This can be stated as shown in Example 18 by assuming the existence of `TotalFlytime` property of a duty as defined in Example 15.

Example 18 One easy duty in three consecutive duties

```
PROPERTY easyDuty OF Duty
  CONSTANTS:
    maxFlytimeInEasyDuty = 4:00;
  RULE:
    TotalFlyTime <= maxFlytimeInEasyDuty
ENDPROPERTY

CONSTRAINT oneEasyDutyInThreeDuties OF Duty
  STATUS: ON;
  RULE:
    FOR EACH d1 -> d2 -> d3 IN ELEMENTS
      (easyDuty OF d1) OR (easyDuty OF d2) OR (easyDuty OF d3)
ENDCONSTRAINT
```

`time` **operator** ARUS can convert from a `datetime` value to a `time` value using the operator `time`. It simply drops the date part in the given `datetime` value. For example `time(03.02.2011 01:30 PM)` is `01:30 PM`. A possible use of this operator is explained in Example 19. Suppose that a flight is called a red-eye flight if it departs between 11:00PM and 05:00AM.

Example 19 Definition of a red-eye flight

```
PROPERTY redeyeFlight OF Flight
  RULE:
    (time(departure_time) <= 05:00AM)
         AND (time(departure_time) >= 11:00PM)
ENDPROPERTY
```

Timewindow Expressions Airline crew pairing problems sometimes require to state a constraint that needs to be checked over a timewindow. For example, "total fly time in a pairing must not be greater than 8 hours in any 24 hour window" is such a constraint. ARUS provides timewindow expressions to enable the description of such constraints. The time window can be stated on three different time units: hour, day, week. General format for this expression starts with keyword `FOR TIMEWINDOW OF` followed by an integer for specifying the width of the time window and then the time unit that is to be used when shifting the window. This is followed by a boolean expression which is the value to be checked. An example is given in Example 20.

As for the priority and the associativity of these operators, we follow the convention of general purpose programming languages. All binary operators are left associative and Table 3 gives the priority of the operators in decreasing order.

Example 20 Time Window Expressions

```
CONSTRAINT the_8_in_24_Rule OF Duty
  STATUS: ON;
  RULE:
    FOR TIMEWINDOW OF 24 hour
      SUM OF flightTime OF ELEMENTS <= 8:00;
ENDDONSTRAINT
```

Priority	Operators
1	unary operators
2	*, /
3	+, -
4	<, <=, >= , >=
5	==, !=
6	AND, OR

Table 3: Priorities of the operators in decreasing order

3 Compiler Design and Implementation

To show and test the effectiveness of our rule modeling language, we designed and implemented a compiler for ARUS. A compiler is a program that reads the source code of a program written in one programming language (called the *source language*) and translates this to an equivalent program in another language (called the *target language*). The source language in our case is ARUS and the target language is C++. The C++ code generated by ARUS compiler is then compiled into a DLL (dynamic link library). This DLL exports methods to check the feasibility and to compute the cost of activities.

We will now give details of ARUS compiler. We start by giving details about the overall system and how ARUS compiler integrated with it in Section 3.1. After that the structure of ARUS compiler is detailed in Section 3.2 and continue with phases of ARUS compiler. We give general information about `lexical` and `syntax analyzer` with details of our implementation in Section 3.3. After that in Section 3.4 we give details about `semantic analyzer` and finally we give the detailed structure of our `code generator` in Section 3.5.

3.1 Airline Crew Pairing Engine and ARUS Compiler Integration

In this section, we will give some details about how ARUS compiler and airline crew pairing engine are integrated to work together.

Our main goal is to separate airline crew pairing engine and feasibility/cost calculations so that users can change the rules and regulations with-

out having to modify the airline crew pairing engine. To ensure this goal, we generate a DLL file from the generated C++ program, to be used as a feasibility checker and cost calculator by airline crew pairing engine. The methods exported by this DLL library and they are used at run time by the airline crew pairing engine as needed.

Since the airline crew pairing system uses the generated DLL file at run time, the data structures used by the engine and DLL library must be the same. In other words, both the DLL file and the airline crew pairing engine use the same activity hierarchy (i.e. flights are the basic activities, duties are formed from flights, pairings are formed from duties, and finally solutions are formed from pairings). The basic activities are given to the engine as an input. However the derived activities are built by the engine. To build a duty for example, the engine starts with a single flight. A single flight forms what we call a *partial duty*. The engine then attempts to extend partial duty by appending another flight. Each time such an extension is attempted, the engine checks the feasibility of the extension. In other words, the engine finds out if this extended form is still a duty or not by using the methods exported by the DLL. This scheme is the same for generating pairings as well. A single duty forms a *partial pairing*. Partial pairings are extended by appending duties, and every extension attempt is checked for feasibility by using the exported DLL methods.

The overall structure of airline crew pairing engine and ARUS compiler is given in Figure 3.

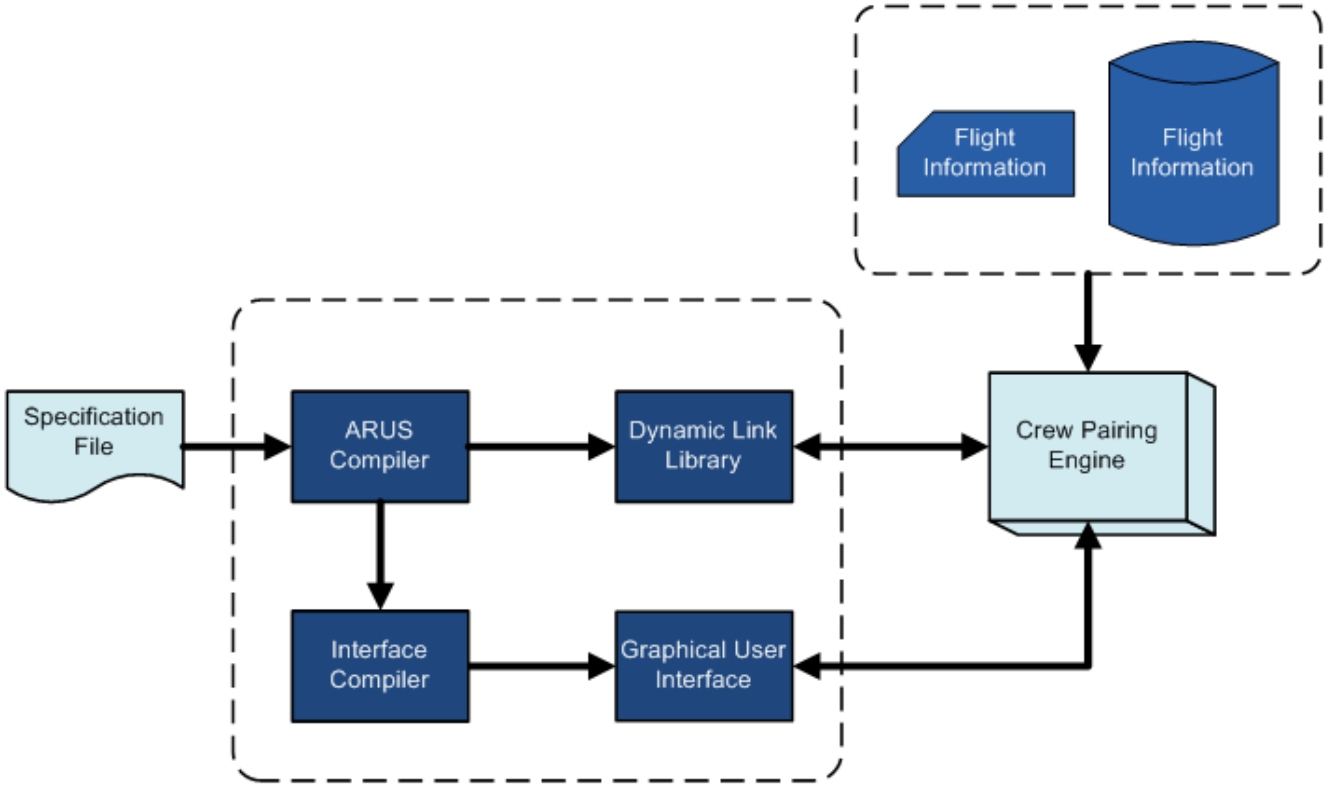

Figure 3: Overall System

3.2 Compiler Structure

In this section we will give the general structure of ARUS compiler. Like general compiler implementations, the implementation of ARUS compiler is composed of two parts: *analysis* and *synthesis* [11].

In the analysis part, the source code is processed according to the grammatical structure of ARUS to generate the intermediate representation of the program with necessary information produced from the source code. This part is generally composed of several phases: *lexical analyzer*, *syntax analyzer* and *semantic analyzer*.

In the synthesis part, the generated representation and gathered information is used to generate corresponding translation of the source language to the destination language. There are intermediate phases where the generated representation is processed further before generating the destination language. In ARUS compiler, these processes are done in *code generation* phase.

We give a detailed structure of ARUS compiler in Figure 4. The phases given in this figure will be explained in the following sections.

3.3 Lexical and Syntax Analyzer

The lexical analyzer, also called scanner, is the first phase of the analysis part of a compiler. It takes characters of the source program code as input and generates a sequence of tokens (sequence of characters that have a collective meaning) which is used by the syntax analyzer [12].

The syntax analyzer, also called parser, is the second phase where the

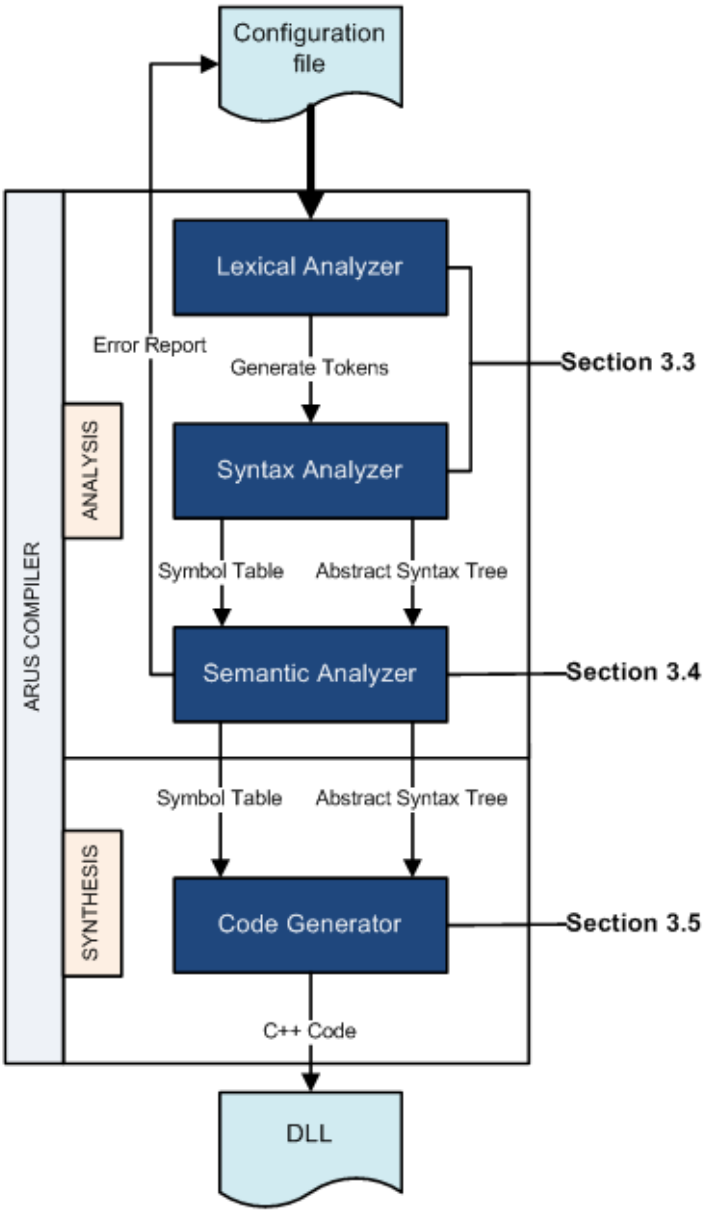

Figure 4: ARUS Compiler Structure

tokens from the lexical analyzer are verified according to the grammar of the source language [12]. It takes the sequence of tokens generated by the lexical analyzer and converts it into a data structure that will be used in further processing.

For generating these analyzers, we used *flex* (Fast Lexical Analyzer) and *bison* (GNU parser generator) [13]. We now give a brief information about these tools.

flex is a scanner generator designed for lexical processing of the character streams. Basically, flex takes a set of regular expressions specifying the tokens of the source language. It generates a program for recognizing these regular expressions from an input stream. The actions associated with the regular expression are used to generate the tokens corresponding to these regular expressions.

bison is a general purpose parser generator that takes an annotated context free grammar description of the source language. It generates a "Look Ahead Left-to-Right Rightmost" C or C++ parser which can parse a sequence of tokens that conforms the grammar. It is mostly used with Flex which generates corresponding tokens from input streams.

With the help of generated lexical and syntax analyzers, we process the source code and generate intermediate data representations to be used in latter phases. These representations are general data structures that is used in compiler implementations:

Parse Tree A *parse tree*, or a *concrete syntax tree*, is a rooted tree representation of the syntactic structure of the input according to the grammar.

Non–leaf nodes are labeled by non–terminals and the root is labeled by the start symbol. The leaves are labeled by terminals.

Abstract Syntax Tree An *abstract syntax tree* (AST) is a representation of the input as a tree. It is different than a parse tree in that a parse tree give the concrete syntax structure of the input, whereas in an abstract syntax tree, the concrete syntactic details do not exist. The interior nodes represents the programming constructs rather than nonterminals.

Symbol Table A *symbol table* serves as a database for compilation process [12]. They are designed as data structures to hold information about the source program constructs. The main contents of the *symbol table* are the type and attribute information of each user–defined identifier in the program. The information is collected incrementally by the lexical and syntax analysis phases and used in latter phases like semantic analysis and code generation phases to get needed information about the identifiers in the source program.

3.4 Semantic Analyzer

In this phase, the source program is checked for semantic consistency by using the abstract syntax tree and the symbol table. We have to ensure that the input program is semantically correct to continue for code generation.

Semantic analyzer of ARUS compiler uses *multi-pass* approach where the abstract syntax tree is processed more than once. It gets the abstract syntax tree and symbol table generated by the previous phases and traverse them to check for the semantic validation rules.

Semantic validation rules are mostly based on tracking declaration/ definition consistency. As each of these rules are checked separately, the abstract syntax tree and the symbol table is traversed more than once to generate corresponding error and warning messages. We now give further explanation about the semantic validation rules.

Undeclared definitions This validation rule checks if all the definitions (property or constraint) given in definition part are declared within an activity declaration in the declaration part. This check is performed by using the symbol table. Each identifier used in the source program has a record in the symbol table. For this check, we traverse thw symbol table for any unmatched definitions and give corresponding errors with line numbers.

Undefined declarations Like the previous validation rule, this rule checks if all declarations (property, constraint) given within an activity declaration in the declaration part have a definition in the definition part. Again the symbol table is used to perform this check. Corresponding error messages are produced if an undefined declaration is seen.

Use of undeclared identifiers This validation rule checks if there are any identifier that are used without a record in symbol table. Each identifier used in a rule expression has to be declared beforehand. The identifiers in an expression can be a constant, a variable, or an entity (property or constraint) identifier declared within an activity declaration. Each identifier in an expression are checked for a declaration from the symbol table and error messages are given if there is an undeclared identifier

used.

Supported operations This validation rule checks if all the operators in expressions are valid operators. The supported operators and their resulting types are given in Table 1 and Table 2. Each operator is checked from these tables and corresponding error messages are produced if necessary.

Table expression check This validation rule checks if the table declarations are semantically correct. The table declarations are given with declared variable names for the predicates of rows and columns. In Example 21, the variable names for the predicates are declared to be `numOfLegs` and `startTime` whereas the variables used in the predicates are `otherStr` and `anotherStr`. So given example will fail this validation and will generate an informative error message for variable name mismatch.

Example 21 Table Variable Mismatch

```
TABLE maxDutyTime
 numOfLegs/startTime  | 06:00AM<= otherStr <03:00PM | OTHERWISE
 1 <= anotherStr <= 2 |             14:00           |   12:00
 3 <= anotherStr <= 5 |             13:00           |   10:00
```

Type checking This validation rule is one of the most common checks performed by compilers. It ensures that each operation performed in the program respects to the type system of the language. In ARUS, types

of each property is given in the declaration part. To ensure the correctness of type assignments, we look if all properties and constraints, declared in the declarations part, have an expression of the same type.

Example 22 is a basic example of type mismatch. The property *maxDuration* of *Duty* is declared as a duration type, but the rule that is defined for `maxDuration` computes an integer value. We check such cases by finding the type of each expression with respect to Table 1 and ensure that the computed types are the same as the declared types.

Example 22 Type Mismatch

```
ACTIVITY Duty
  ...
  PROPERTIES:
     ...
     maxDuration : duration;
     ...
ENDACTIVITY

PROPERTY maxDuration OF Duty
  ...
  RULE:
      10 * 2;
   ....
ENDPROPERTY
```

3.5 Code Generation

In this section, we go into details of the final phase of ARUS compiler, *code generation*. This phase takes the abstract syntax tree and the symbol table from the previous phases, and produces a C++ program implementing the feasibility and cost calculation rules given in the input. The generated C++ code is then used to generate a DLL (dynamic link library) for performing feasibility/cost calculations. Finally generated DLL file is used by the airline crew pairing system at run time.

The source program in ARUS, is a specification of activity based rules and regulations which are used in feasibility/cost calculations of airline crew pairing engine. To generate equivalent methods in C++, we represent each defined activity type (e.g. flight, duty etc.) as a class and their corresponding property and constraint definitions as methods of this class.

We will explain the code generation phase of ARUS compiler. We start with code generation of constant declarations in Section 3.5.1. After that we detail the code generation of activity declarations in Section 3.5.2. Finally, the code generation for property and constraint definitions are described in Section 3.5.3 and feasibility checking methods in Section 3.5.4.

3.5.1 Code Generation for Constants

Constant declarations can be made either in global constants declaration section of the specification or within entity (property or constraint) definition section. The constant declaration semantics of ARUS is very similar to the constant declaration in C++. So the code generation for constant declarations is very simple. We just add the data type for the constant declaration

of build–in data types. On the other hand, constructors like `set` or `table` has more complex code generation methods. We use array structure of C++ to represent these constructors. `Set`s and `sequence`s are represented as one dimensional arrays whereas `table`s are represented by two dimensional arrays in generated C++ code.

The constants given in global constants declaration section, are used to generate a header file which will be included by all the generated activity classes. Below are the examples of a global constant declaration section and its corresponding generated C++ header file.

```
CONSTANTS:
    IstanbulAirports = {SAW, IST};
    briefingTime = 00:30;
    debriefingTime = 00:45;
ENDCONSTANTS
```

```
//Declerations.h
//Global Constants
airport IstanbulAirports[] = {SAW, IST};
duration briefingTime(00:30);
duration debriefingTime(00:45);
```

Other than the global constant declaration section, constants can be defined in the entity (property or constraint) definitions. These constants simply generated as C++ local constants specific to the generated method for the defined entity.

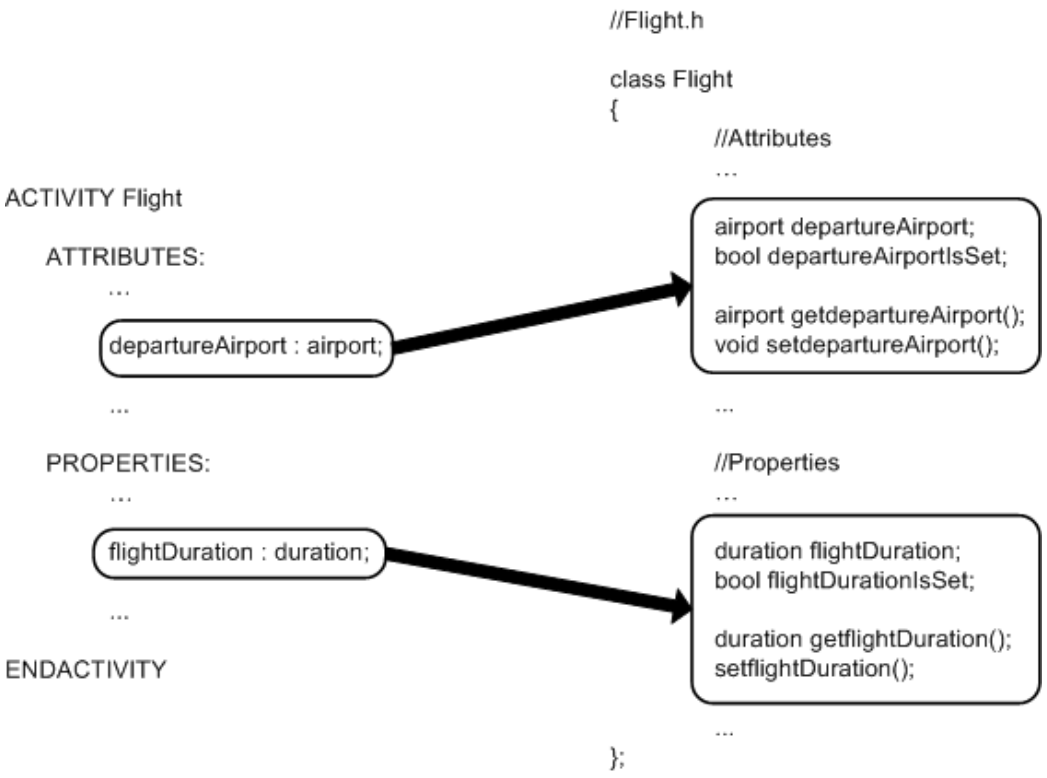

Figure 5: Base Activity Code Generation

3.5.2 Code Generation for Activities

As mentioned in earlier sections, an activity is represented by a class in C++. Activities are composed of attributes, properties or constraints. Likewise C++ classes are composed of attributes and methods. Each entity (attribute, property or constraint) declared in an activity corresponds to an attribute in the generated C++ class. We give examples of generated class declarations for basic and derived activity declarations in ARUS in Figure 5 and Figure 6.

As seen in Figure 5, the `Flight` activity declaration consists of attribute and property sections. These declared attributes and properties are directly

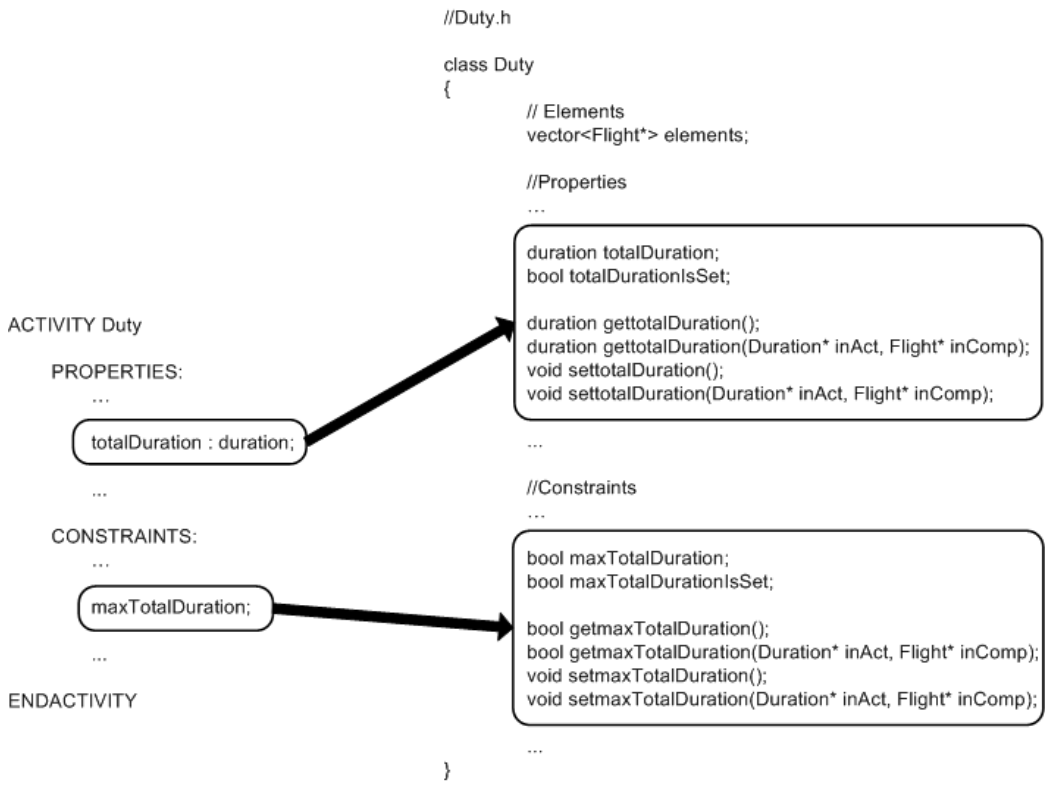

Figure 6: Derived Activity Code Generation

used as attribute declarations in the generated class with their setter and getter methods. Along with the getter-setter methods, a boolean flag `IsSet` is declared. This flag holds the information if the attribute is set or not.

Derived activities are composed of a sequence of other activities. They also have properties and constraints. Different than basic activities, we generate two overloaded setter functions for each entity (property or constraint). Another difference of the generated class declaration of derived activity given in Figure 6, is that it contains an additional attribute of vector type, `elements`, which holds the sequence of composing activities.

3.5.3 Code Generation for Entity Definitions

In this section, we give some details about the entity (property, constraint) definitions and their representation in the generated C++ code. We said that an activity declaration with its properties and constraints, is represented as a class. A property or a constraint of an entity is implemented by an attribute of this class together with getter and setter methods for this attribute.

The generated getter methods have a common easy implementation for each property and constraint definition. It first checks if the attribute is set before. If not it calls the setter method, and then returns the value of the attribute. We give an example of getter functions in Method 1.

The setter methods have a more complex structure as they are used for calculating the property and constraints. The calculation methods are based on expressions given in the definition of the entities. Each expression corresponds to a different method generation. We now give details about generated methods for each expression type:

Method 1 Getter Methods

```
duration Duty::gettotalTime()
{
    if(!totalTimeIsSet)
        settotalTime();

    return totalTime;
}
duration Duty::gettotalTime(Duty* inAct, Flight* inComp)
{
    if(!totalTimeIsSet)
        settotalTime(inAct, inComp);

    return totalTime;
}
```

Atomic Expressions As atomic expressions correspond to simple expressions, the code generation method for them is very simple. Each atomic expression corresponds to a constant value (e.g. 7, SAW etc.) or an identifier of a constant or an entity (attribute, property or constraint). The generated code for these expressions is based on the type of the expression. If expression is a constant value or a constant identifier, their representation is same with ARUS representation. On the other hand, if expression is an identifier of an entity than it is represented as the value of the corresponding entity by using the generated getter method.

Arithmetic Expressions As ARUS supports only the basic arithmetic operators that all general purpose programming languages support, we do not have to generate an explicit code generation procedure for it. In Method 2, we give the code generated for the property definition given in Example 11. Duration of a flight is calculated by simply subtracting the `departure_time` of the flight from the `arrival_time`. As the identifiers used within the arithmetic operation, corresponds to an attribute of flight activity, they are represented as constant variables that are set using by the getter methods of those attributes.

Boolean Expressions ARUS supports common boolean expressions (NOT, AND, OR) that are also supported in C++ but with different representations. The code generation procedure for a boolean expression is simply to replace supported operators with C++ equivalent. In Table 4, we gave each boolean expression with its C++ equivalent.

Method 2 Calculation Method for Arithmetic Expressions

```
void Flight::setflight_duration()
{
    duration result(00:00);
    time tempVar1 = getarrival_time();
    time tempVar2 = getdeparture_time();
    result = tempVar1-tempVar2;
    flight_duration = result;
    flight_durationIsSet = true;
}
```

Boolean Expression	C++ Equivalent
NOT	!
AND	&&
OR	||

Table 4: ARUS Boolean Operators, C++ equivalents

Relational Expressions Relational operators in ARUS is the same as the relational operators in C++. Therefore the generated code for these expression are the same as the input expression. One particular case where ARUS relational expressions differ from C++ is the chained relational expression. The chained relational expressions are represented as a sequence of simple relational expressions which are connected to each other by an `AND` operator. For example, `1 <= x <= 5` is translated as `(1 <= x) AND (x <= 5)`

Set Membership Each `IN` or `NOT IN` operator, performs a membership test. The sets in ARUS are implemented by one dimensional arrays in C++. Set membership checks are simply implemented by a code that goes through all the elements in the array sequentially.

```
IstanbulAirports = {IST, SAW};

arrivalAirport IN IstanbulAirports;
```

```
bool isInIstanbulAirportsSet(airport inAirport)
{
    for(int i = 0; i<IstanbulAirports.size(); i++)
    {
        if(IstanbulAirports[i]==inAirport)
            return true;
    }
    return false;
}
```

Conditional Expressions ARUS uses the same syntactic representation of conditional expressions as C++. Given an expression like;

```
IF (cond) e1; ELSE e2;
```

The generated code will be the same except, the use of C++ syntax with generated expressions.

Table Reference A table is represented as a matrix of constant values. For referencing a table, we generate a function to calculate the row and the column for the reference. After the row and the column for the reference are calculated, then we simply refer to the corresponding value in the matrix for this row and column. The function calculating the row and the column indices checks the table predicates. The input and output types of the generated method is determined by types used in the predicates and types of the cell values. In Method 3, we give an example of generated method for referencing the table given in Example 1.

Time Window Expressions Time spanning rules are common in airline domain. This expression type performs calculations over a time window. These expressions require additional functions to be generated by the compiler as the whole activity will be traversed and the expression is calculated for each time window. Below we give an example of generated function for time window expressions.

In the generated function, the elements are traversed in by sliding windows according to the declared window size and step. A partial *Duty* is generated by the function `getWindowOf` which takes a sequence of composing activities, *Flight*, with the time span coefficient. The corresponding expression is checked and the elements trimmed from the beginning to shift the window one unit. This unit can be hour, day, or week. This process is repeated until the end of the elements is reached. If expression fails in one of the iterations, the loop is broken and the result is set to be false.

Method 3 Calculation Method for Table Reference

```
duration getMaxDutyTime(int numOfLegs, Duration startTime)
{
    int row, column;
    if( 1<=numOfLegs && numOfLegs<=2)
        row = 0;
    else if(  3<=numOfLegs && numOfLegs<=5)
        row = 1;

    if( 06:00AM<=startTime && startTime<03:00PM)
        column = 0;
    else
        column = 1;
    return maxDutyTime[row, column];
}
```

Method 4 Generated time window calculation function

```
void Duty::setmaxFlightTimeTW()
{
    bool result = true;
    vector<Flights*> tempList = elements;
    int timeSpan = 5;
    Duty* tempAct = getWindowOf(tempList, timeSpan);

    while(tempAct != NULL)
    {
        result = tempAct->getmaxFlightTime();
        if(!result)
            break;

        trimElements(tempList);
        tempAct = getWindowOf(tempList, timeSpan);
    }

    maxFlightTimeTW = result;
    maxFlightTimeTWIsSet = true;
}
```

Special Operations ARUS supports some special operations (e.g. SUM OF, AVG OF, FIRST OF etc.) which can be applied on different type of sets or sequences of activities. Each operator has its own C++ representation. In Method 5, we give an example of generated setter method for the property defined in Example 15.

Method 5 Generated setter method for totalFlyTime

```
void Duty::settotalFlyTime()
{
    duration result(00:00);
    duration tempVar;
    for(int i = 0; i<elements.size() ; i++)
    {
        tempVar = elements[i]->getflight_duration();
        result += tempVar;
    }
    totalFlyTime = result;
    totalFlyTimeIsSet = true;
}
```

In the example, `totalFlyTime` is calculated by summing up the `flight_duration` of each of the `ELEMENTS`. The generated method simply has a loop for adding the values of `getflight_duration()` for each element. `ELEMENTS` keyword corresponds to the elements of the activity, and it is represented as a vector of composing activities.

For the expressions that uses these operators over `ELEMENTS` of a derived activity, we generate an overloaded setter method. This method calculates properties by using the calculations that is done on previous partial activities. In Method 6, we give the generated overloaded setter method for the same property definition of Method 5. Please recall that pairing generator works in an incremental way. A partial derived activity is attempted to be extended by adding a composing activity. The overloaded method avoids performing the same computation over and over, by simply reading the previously computed value of the partial derived activity.

Method 6 Generated overloaded setter method for totalFlyTime

```
void Duty::settotalFlyTime(Duty* inAct, Flight* inComp)
{
    duration result(00:00);
    duration tempActVal = inAct->gettotalFlyTime();
    duration compVal = inComp->getflight_duration();
    result = actVal + compVal;
    totalFlyTime = result;
    totalFlyTimeIsSet = true;
}
```

For `SUM OF` operator, the generated overloaded setter method simply gets `totalFlyTime` of the partial activity (Duty) and and adds `flight_duration` of the composing activity.

Other operators (e.g. AVG OF, COUNT OF etc.) has a similar representation in C++. For example, `MIN OF` operator finds the minimum of a value over a set or sequence by simply comparing value of each element. Again if this operator is used over a property of `ELEMENTS` of a derived activity, overloaded setter method will simply compare the calculated minimum value of partial activity with the property value of composing activity used in the expression.

3.5.4 Feasibility Checker

The feasibility of an activity is determined by checking all the constraints that are defined for that activity declaration. If all the constraints are checked to be true than the generated activity is feasible.

For each activity defined in the specification program, we generate a feasibility checking method that is used by the engine. The engine considers partial derived activity and a composing activity and asks if this partial derived activity can be extended by this composing activity. Our feasibility checking method therefore takes partial derived activity and a composing activity as input parameters and checks if the given composing activity can be added to the given partial derived activity without violating a feasibility rule.

4 Conclusions and Future Work

In this thesis, we designed a high level 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 [10]. ARUS specializes DRL in a domain specific manner. Using such a high level language for the specification of feasibility and cost calculation simplifies the description of these rules.

We also implemented a compiler for ARUS to translate ARUS specifications to C++. The C++ code generated has methods to check the feasibility of derived activities and it also has methods to calculate the costs of the activities. A pairing generator can import these methods and use them for checking feasibility and calculating cost. By separating the feasibility and cost calculation from the pairing generator, it is no longer necessary to change code of a pairing generator to modify the feasibility check and cost calculation.

One other important advantage of ARUS is that users do not have to have a programming experience. We also believe that the syntax of ARUS is simple enough, a little training will be sufficient for novice users. For comparison with the other languages used for the same purpose, we haven't carried out a usability study, but below we give one example rule represented both in Carmen Rave language in Rule 1 and in ARUS in Rule 2. As Rave uses a general purpose programming language like representation, it is more difficult to understand for the end-users who do not have any programming experience in general purpose languages.

As a future work, we plan continue adding new functionality to the lan-

Rule 1 Rave representation for maximum number of international flights in a duty

```
/*
** Rule
** Max number of domestic to international flights per duty
*/
rule (on) max_num_leg_dom_to_int_check =
  valid trip.%check_rules%;
  %num_dom_to_int% <= %max_num_legs_dom_to_int%;
  remark "Max number of legs, in a duty, from domestic
         to international check";
end


%max_num_legs_dom_to_int% =
  parameter 1 minvalue 1
  remark "Max number of legs, in a duty, from domestic
          to international";

%num_dom_to_int% = count(leg(duty))
                            where (%leg_is_dom_to_int%);

%leg_is_dom_to_int% = leg.%departure_is_domestic%
                      and leg.%arrival_is_international%;
```

Rule 2 ARUS representation for maximum number of international flights in a duty)

```
PROPERTY flight_status OF Flight
  RULE:
    IF (departure_airport IN DOMESTIC_AIRPORTS) AND
     (arrival_airport NOT IN DOMESTIC_AIRPORTS)
    "INTERNATIONAL_FLIGHT"
    ELSE
    "DOMESTIC_FLIGHT"
ENDPROPERTY

CONSTRAINT maxNumOfIntFlights OF Duty
  COMMENT: Maximum number of flights from domestic
           to international cannot be more than 1;
  CONSTANTS:
    maxFlights = 1;
  RULE:
    COUNT OF ELEMENTS WHERE
      (flight_status != "DOMESTIC_FLIGHT") <= maxFlights;
ENDCONSTRAINT
```

guage and compiler implementation. We plan to add code generation for newly added expressions (`FOR EACH` and `time`). Also the tables can be extended to have more than two variables. Users can give any number of predicate to be checked for a table. Another future work can be the implementation of a user interface where users can modify constant values that is used in property or constraint calculation methods. With user interface, users can directly change values without having to change the ARUS source. Also further improvements on the run time of feasibility checking procedures can be made.

References

[1] R. Galia and C. Hjorring, "Modelling of complex costs and rules in a crew pairing column generator," in *Operations Research Proceedings 2003*, pp. 134–140, 2003.

[2] C. A. Hjorring, S. E. Karisch, and N. Kohl, "Carmen systems' recent advances in crew scheduling," in *Proceedings of the 39th Annual AGIFORS Symposium*, pp. 404–420, 1999.

[3] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, pp. 316–344, 2005.

[4] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, pp. 26–36, 2000.

[5] W. Taha, "Plenary talk iii domain-specific languages," in *Computer Engineering Systems, 2008. ICCES 2008. International Conference on*, pp. xxiii –xxviii, 2008.

[6] 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.

[7] 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*, pp. 286–292, 1998.

[8] 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*, pp. 821–830, 1995.

[9] C. Goumopoulos and E. Housos, "Efficient trip generation with a rule modeling system for crew scheduling problems," *Journal of Systems and Software*, vol. 69, no. 1-2, pp. 43 – 56, 2004.

[10] K. X. Thrampoulidis, N. Diamantopoulos, and E. Housos, "Redom: An oo language to define and on-line manipulate regulations in the resource (re)scheduling problem," *Softw., Pract. Exper.*, vol. 27, no. 10, pp. 1135–1161, 1997.

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison Wesley, 2007.

[12] R. W. Sebesta, *Concepts of Programming Languages*. Addison Wesley, 2001.

[13] J. Levine, *flex & bison*. O'Reilly, August 2009.

A Some Real Life Examples

In this section, we give some real life rules and their representation in ARUS.

Rule 3 Maximum flight time of a duty is 5 hours if it starts from a base airport, else it is 4 hours.

Rule 3 puts a limit on total flight time of a duty, depending on the starting airport of the duty. To check this feasibility rule, we have to calculate the total flight time in a duty. The first step is to calculate the flight time of a single flight. Since a duty is a sequence of flights, we can then simply sum the flight time of all the flights in the duty to find the total flight time in a duty. The first two properties in Rule 3 perform these calculations. Since we are also interested in the starting airport of a duty, we also write a property to get the departure airport of the first flight in the duty.

Finally the constraint in Rule 3 compares the total flight time computed by the property to the required limit, where the limit changes according to the starting airport of the duty.

Rule 4 Minimum sit time between two consecutive flights in a duty can not be less than 30 minutes.

In Rule 4, sit time between two consecutive flights is required to be more than allowed minimum sit time. This rule can be given as a single constraint in ARUS. The constraint considers each consecutive flight pair `f1` and `f2` in the elements of `Duty`. It subtracts the `arrival_time` of the first flight from the `departure_time` of the second flight (which gives the sit time between these flights) and checks if the result is more than `minSitTime` which is set to be 30 minutes.

Rule 3 Maximum Flight Time in a duty

```
PROPERTY flightTime OF Flight
  RULE:
    arrivalTime - departureTime;
ENDPROPERTY
PROPERTY totalFlightTime OF Duty
  RULE:
    SUM OF flightTime OF ELEMENTS;
ENDPROPERTY
PROPERTY startingAirport OF Duty
  RULE:
    departureAirport OF FIRST OF ELEMENTS;
ENDPROPERTY
CONSTRAINT maxFlightTime OF Duty
  STATUS: ON;
  CONSTANTS:
   BasePorts = {SAW, IST, ADA};
  RULE:
   IF startingAirport IN BasePorts
     totalFlightTime < 05:00;
   ELSE
     totalFlightTime < 04:00;
ENDCONSTRAINT
```

Rule 4 Minimum sit time between two consecutive flights of a duty

```
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
```