UNIVERSITY OF GOTHENBURG

Encoding of the contract language CL into the Grammatical Framework (GF)

Seyed Morteza Montazeri

Bachelor of Applied Information Technology Thesis

Report No. 2010:016
ISSN: 1651-4769

Academic Supervisor: Gerardo Schneider

University of Gothenburg
Department of Applied Information Technology
Gothenburg, Sweden, May 2010

Encoding of the contract language $\mathcal{CL}$ into the Grammatical Framework (GF)

Seyed Morteza Montazeri
shayanmont@gmail.com

IT University of Gothenburg, Software Engineering and Management. Gothenburg, Sweden

CHALMERS | GÖTEBORGS UNIVERSITET

Abstract

* * * * * * * * * * *

Consistent requirement adoption is important in almost all of the software engineering projects. The demand of conflict free requirements, requires a way to manage requirements in more reliable way which means to detect conflict formally. In this paper, we describe the foundation and implementation of a tool that enables the requirements written in natural language (English) to be represented in a formal language ($\mathcal{CL}$). This in effect enhances the possibility to apply CLAN (a tool which analyzes requirements or contracts written in $\mathcal{CL}$) for analysis. We have also applied our approach to a case study where the findings show correct detection of conflicts in requirements.

Keywords: Grammatical Framework, Requirements, Contract, Conflict analysis, Natural language

1 Methodological Aspect

1 Introduction

THIS project will make an attempt to demonstrate the application and implementation of tool written in Grammatical Framework (GF: programming language for multilingual grammar applications (Ranta, A., 2009)) to enable the translation back and forth between restricted natural language and a formal language $\mathcal{CL}$. This is a technical project, whereby, we also try to demonstrate the application of the prototype on a case study to achieve and show the anticipated results.

In the next section we give background of technical terms used as basis of this research. section 3 reports on classic transfer based system and its comparisons with our system. section 4 is where the implementation is discussed. In section 5 we evaluate our approach by using a case study. In section 6 the results of the research and case study are illustrated. The paper is rounded off with sections on related work, conclusion (where conclusion together with future work are addressed) and acknowledgements.

Background

Requirements engineering is a process designed to produce set of clear, consistent and complete requirements by iterative discovery and analysis of requirements. This process is usually hard and difficult to manage. Set of newer requirements are sometimes inconsistent and in contradiction with the set of requirements that were already set. The introduction of these inconsistencies in requirements usually may lead to violation of what is stated as goals of development (Robinson and Pawlowski, 1999).

Requirements conflict and inconsistency analysis thus is one of the most vital tasks that must be considered in a software engineering project and generally in companies where new set of requirements are aggregated and accumulated in a time period. In this time period, while set of new requirements are added, the old ones are also updated besides the fact that these lead to development of the requirements document overtime (Robinson and Pawlowski, 1999). However, the document in this situation is usually in a transitory state where many semantic conflicts exist since one change may result in myriads of other changes (Robinson and Pawlowski, 1999). On the other hand, most of these conflicts could be solved if understood by the analysts, though, most of the time they remain in the document and are only detected late in the development process.

More and more companies such as Saab AB are challenging with the problem of identifying inconsistencies and conflicts in requirements. Challenges mentioned earlier lead to the need for some kind of automation that facilitates the checking of requirements. Much research has been invested into the usage of formal language such as the one stated in (Prisacariu and Schneider, 2009) which is designed for analysis of contracts, requirements contract (RC, 2009) or any negotiation process such as Internet-based negotiation and contracting in the e-business and e-government environments. However, it shouldn't be forgotten that requirements and requirements contracts are still written in NL (natural language) (Hahnle et al., 2002). Although NL is typically used for communication and can provide powerful understanding for human, it is not appropriate for analysis. As a result, there is a need for some kind of formal languages that can provide suitable ways for analysis purpose. The contract language $\mathcal{CL}$ is a logic (Prisacariu and Schneider, 2009) which can help the process of analysis and automation of contracts: detection of contradictions and inconsistencies, identification of superfluous clauses and checking some desired properties on a contract. At last, this can help to process the NL which the requirements are written if presented in a CL. In other words, we are aiming at building a system or program to be able to give automatically the correspondent $\mathcal{CL}$ formulas of the NL and vice versa. In this case we would be able to analyze these formulas for detecting inconsistency.

Problem statement

The Case: Saab AB
"Saab AB serves the global market with world-leading products services and solutions ranging from military defense to civil security (Saa, 2010)". One of the challenges Saab AB is facing concerning requirements is how to effectively detect and solve conflicting requirements. During the development phases, inconsistencies and conflicts in the requirements are usually detected by reviews of requirements. This process usually takes place early in development phases. Nevertheless, designers and developers also provide feedback in case of detection of any inconsistency during the development phases to a third party for further fixes. They are also looking at model based engineering in which the requirements are replaced with models. In this case any inconsistencies are to be found when reviewing the model, but this process is really difficult to automate (Berling, 2010).

In this paper we suggest solutions to the problems and challenges just mentioned. We show that it is possible to define a connection between source language in our case $\mathcal{CL}$ and specification language which is natural language (NL) and thus to be able to write the requirements in a formal language like $\mathcal{CL}$. This in effect, enhances the possibility to seek and detect inconsistencies in the requirements by further analysis that could be conducted in CLAN. This process however, should be back and forth, in other words, the translation back could be done for two reasons: (i) if the requirements are written in a formal language like $\mathcal{CL}$ and we want to have a user friendly visualization (ii) if you use $\mathcal{CL}$ for conflict detection and want to show the result in NL. We will concentrate on CL as a source language but have in mind NL as specification language. Our approach is based on the Grammatical Framework (GF) (Ranta, A., 2009),"flexible mechanism that allows to combine linguistic and logical methods" (Hahnle et al., 2002). The main idea is first to specify an abstract syntax for a source language (in our case $\mathcal{CL}$), and second, to specify concrete syntaxes and thus map them to NL.

Purpose

The purpose of this paper is to investigate ways to automate the process of conflict detection in requirements. The scope of the automation in this paper is the formal language $\mathcal{CL}$ and the implementation language GF (Ranta, A., 2009) which both have a predominant role in this research. The specific research questions addressed in this paper are (i) the solution to map CL and its logical formulas to the abstract syntax of Grammatical Framework primarily and thus try to find how it is possible to translate full CL to abstract syntax and vice versa? and (ii) how it is possible to translate abstract syntax to concrete syntax and from there to natural language and vice versa? The motivation for doing this study stems from the fact that more and more companies would be able to deal with requirements in more efficient and faster way and thus confining to high standard requirement. The result is an automation for requirements that supports the detection of conflicts and inconsistencies in the requirements and thus a change from a natural language into controlled language without any conflicts.

2 Methods

Finding an appropriate methodology was not easy, given that the goal of this research does not follow numerical analysis or interview based directions researches (Osterwalder, 2004). However, like many other projects, which their main focus is on technical aspects, Design Science (DS) paradigm will be used to address the needs of this research. DS paradigm boosts to a large extent the capabilities within organization by mapping what is considered to be a practical needs in to specific research interest and thus creating new and innovative artifacts (Hevner et al., 2004). However, these artifacts are not necessarily full-grown systems than can be used in practice but rather the innovation which defines ideas (Hevner et al., 2004). DS was chosen because of its influential positive effects in accomplishing wide variety of tasks such as analysis, design, implementation with the use of ideas and practices based on innovations. It seeks to create "what is effective" and not to find "what is true" in comparison with other paradigms (Hevner et al., 2004).

The primary reason for choosing DS is its fundamental paradigm of problem-solving (Hevner et al., 2004) which it stems from the fact that causes humans to abandon the previous problems they encountered (Osterwalder, 2004). " It [DS] is solution-oriented, using the results of description-oriented research from supporting (explanatory) disciplines as well as from its own efforts, but the ultimate objective of academic research in these disciplines is to produce knowledge that can be used in designing solutions to field problems. " (Van Aken, 2005). The other characteristic worth mentioning is the heuristic nature of DS toward its abstraction and representation of means and ends (Hevner et al., 2004). This nature or strategy usually results in designs that are feasible and appropriate and thus as a result lead to the implementation of those designs later on (Hevner et al., 2004). DS research methodology , as a result can be considered as highly effective research methodology for this research in relation to what is provided as a background of this research, and by indicating the possibility of translating full CL to abstract syntax of GF through some kind of automation or artifact.

Philosophically, the arguments regarding design science stems from the pragmatists (Aboulafia 1991) (Hevner et al., 2004) and the objectives of this research methodology are justifiable through pragmatic validity (Van Aken, 2005). Pragmatism arises out of actions, situations, and consequences rather than antecedent conditions (Creswell, 2008). John W. Creswell (Creswell, 2008) also stated that "instead of focusing on methods, [pragmatic] researchers emphasize the research problem and use all approaches available to understand the problem " (see Rossman & Wilson 1985). For this research project that can be considered as a technical one, the pragmatic worldview seems to be an appropriate paradigm.

Data Collection

In this section a detailed information about the process of data preparation and collection is elaborated. Different data was extracted and gathered to facilitate the performance of set of activities such as decision making and knowledge sharing. Mainly, these data was gathered in order to gain information regarding different aspects of this study such as CL, GF and some conceptual ways to map CL to GF. The process of data collection started early in the project and mainly through literature review. In the next section more detail will be provided on literature review.

Literature Review

Primarily, the literature review is used to get a hands on published materials in different subject area of this research and to provide theoretical background needed for this research. Considerable number of articles on different subjects related to research were read including articles related to CL and GF to set the basis ground for understanding the roots. On the other hand, set of other articles were read to get the concept and structure on how to encode CL into GF. Several search engines such as Google Scholar, Chans (Chalmers University's library) were used used to carry out this literature review.

Using literature review for collecting data provided useful information from valuable references to understand specific problems and thus gaining valuable ideas on how to solve the problem. Literature review also falls under pragmatism since in pragmatic world view, the researcher use all approaches to understand the problem (Creswell, 2008) and literature review in our case is one of those possible approaches to understand a problem.

2 Background

1 The Contract Language $\mathcal{CL}$

There are various approaches in defining formal language for contracts and requirements contracts. Some of them focus on the definition of contract taxonomies (Aagedal, 2001; Beugnard et al., 1999), while others provide a formalization based on logics, e.g. classical (Davulcu et al., 2004), modal (Daskalopulu and Maibaum, 2001), deontic (Governatori and Rotolo, 2006) (Paschke et al., 2005), or defeasible logic (Governatori, 2005) (Song and Governatori, 2004). Other formalizations are based on models of computation (e.g. FSMs (Molina-Jimenez et al., 2004) and Petri Nets (Daskalopulu, 2000)).

As stated in (Prisacariu and Schneider, 2007) the best approach in defining formal languages for contracts is based on logic. A logic that contains properties of deontic notions such as obligation, permission and prohibition but not necessarily based on deontic logic.

$\mathcal{CL}$ is a logic based on combination of deontic, dynamic and temporal logics where " It designed to reason about contracts and its goal is to preserve many of the natural properties and concepts relevant to legal contracts, while avoiding deontic paradoxes, and at the same time to have a suitable language for the specification of software contracts" (Prisacariu and Schneider, 2009). With the help of $\mathcal{CL}$ it is possible to represent deontic notions of obligation, permission and prohibition. In the following table the syntax of $\mathcal{CL}$ is defined and the rest of this section is devoted to possible elaboration and explanation on notations and terminology of $\mathcal{CL}$ following(Prisacariu and Schneider, 2009).

$$C := C_O | C_P | C_F | C \wedge C | [\beta]C | \top | \bot \tag{1}$$
$$C_O := O_C(\alpha) | C_O \oplus C_O \tag{2}$$
$$C_P := P(\alpha) | C_P \oplus C_P \tag{3}$$
$$C_F := F_C(\alpha) \tag{4}$$
$$\alpha := 0 | 1 | a | \alpha \& \alpha | \alpha . \alpha | \alpha + \alpha \tag{5}$$
$$\beta := 0 | 1 | a | \beta \& \beta | \beta . \beta | \beta + \beta | \beta^* \tag{6}$$

A contract clause in $\mathcal{CL}$ is usually defined by a formula C and which can be either an obligation (C_O), a permission (C_P) or a prohibition (C_F) clause, a conjunction of two clauses or a clause preceded by the dynamic logic square brackets. O, P and F are deontic modalities, Obligation to perform α is shown by the formula $O_C(\alpha)$ in the table, illustrating the fact that one is obliged to perform α if not, the contract is violated and the reparation contract C must be considered and executed (a CTD). CTDs, or *Contrary-to-Duties*, are exceptional behaviors which can occur if the violation of obligation happens. Prohibition to perform α is shown by the formula $F_C(\alpha)$ in the table, illustrating the fact that one is forbidden to perform α and again if violated the reparation C must be executed (a CTP). CTPs, or *Contrary-to-Prohibitions*, are also exceptional behaviors which can be defined as "the penalty in case a prohibition is violated" (Fenech et al., 2009b). $P(\alpha)$ is interpreted as permission of performing a given action α. It should be noticed that the usage of Kleene star (the * operator) which is representing repetition of actions, inside these deontic modalities is not allowed. Although they can be used in dynamic logic style-conditions. As it is shown in the table, action β is used inside dynamic modality representing a condition in which the contract C must be executed if action β is performed otherwise the contract is satisfied. Binary constructors expressed in the table such as &,.,+ are representing concurrency, sequence and choice in basic actions (e.g. "buy","sell") which together (basic actions and operators) they construct Compound actions, in this case α or β. Conjunction of clauses can be expressed using $\wedge$ operator while between certain clauses the exclusive choice operator ($\oplus$) can be used, $\top$ and $\bot$ are the trivially satisfied (violated) contract. *0* and *1* are two special actions illustrated in the table to represent impossible action and the skip action (matching any action) respectively. The following example is an excerpt from part of a contract between Internet provider and a client, where the provider gives access to the Internet to the client: "The Client shall not supply false information to the Clients Relations Department of the provider" (Pace et al., 2007). If we consider the formalization below:

fi = client supplies false information to Client Relations Department
s = provider suspends service

The $\mathcal{CL}$ representation of above formalizations is as follow :

$$F_{P(s)}(fi) \tag{7}$$

2 Grammatical Framework

"The Grammatical Framework (GF) is a framework for defining grammars and working with them" (Ranta, A., 2009). The history of GF is back to its first implementation in the project called "Multilingual Document Authoring" at Xerox Research Center Europe in Grenoble in February 1998 (Ranta, A., 2009). As explained in (Hahnle et al., 2002) the main idea of this project was to build an editor which helps a user to write a document in a language which is unfamiliar as Italian while at the same time seeing how it develops in a familiar language such as English. However, GF as functional programming language was mainly developed at Chalmers University of technology and Gothenburg University (Ranta, A., 2009). Like most of the compilers, GF has a central data structure called abstract syntax based on Type Theory where special purpose grammars are defined (Ranta, A., 2009). It also has a concrete syntax part which it specifies how the formulas defined in abstract syntax can be translated into a natural language or a formal notation. Some of the applications of GF in the form of prototype has been used in tourist information, business letters and natural-language rendering of formalized proofs. Another example is software specifications where the specifications can easily be defined in abstract syntax (Hahnle et al., 2002). In order to distinguish different kinds of rules, GF has a module system. Basically it has two main modules: the abstract syntax and concrete syntax modules (Ranta, A., 2009).

Abstract syntax

Abstract syntax is a type-theoretical part of GF where logical calculi as well as mathematical theories are allowed to be defined simply by type signatures (Hahnle et al., 2002). Now, let's look at the hello world example below where we define types of meaning which in our case here, are of types `Greeting` and `Recipient` (Ranta, A., 2009):

```
cat Greeting ; Recipient
```

we then define `Hello` as a function for building trees. The type of `Hello` has `Greeting` as its value type and `Recipient` as its argument type. As customary in functional programming languages, argument types and the value type are separated from each other by an arrow (→) (Ranta, A., 2009).

```
fun Hello: Recipient -> Greeting;
```

Concrete Syntax

Once an abstract syntax is constructed, "a concrete syntax can be built, as a set of linearization rules that translates type-theoretical terms into strings of some language " (Hahnle et al., 2002). Linearization basically means generating the language (Ranta, A., 2009). For instance let's look at English linearization rules for the function above :

```
lin Hello recip = {s = "hello" ++ recip.s} ;
```

The `lin` , defines the linearization of `Hello` in terms of the linearization of its argument. This linearization is represented by the variable `recip` although the variable name can be anything such as `x` or `y` . The terminal `''hello''` is in quotes which is concatenated with the `recip.s` using concatenation operator ++, which combines sequence of terminals. The most important thing to consider in a linearization rule is to define a string as a function of the variable it depends on (Ranta, A., 2009).

The example illustrated above shows that the `fun` rules should be defined in the abstract syntax modules and `lin` rules in concrete syntax modules. Abstract syntax is where we need to define powerful type theory to express dependencies among parts of texts and in concrete syntax we need to define "language-dependent parameter systems and complex structures of grammatical objects using them" (Hahnle et al., 2002). At last, the two main functionalities in GF worth mentioning here are linearization (translation from abstract to concrete syntax) and parsing (translation from concrete to abstract syntax) (Hahnle et al., 2002).

3 Conflict Analysis

Generally, there are four main reasons that cause conflicts in contracts or requirement contracts. The first one is when obligation and prohibition takes place on the same action (e.g. obligation to go east and prohibition to go east) in this case performing any action will lead to a violation of the contracts. The second conflict type happens when permission and prohibition takes place on the same action (e.g. permission to pay and prohibition to pay) which will also lead to violation of contracts. The other two cases are obligation to perform mutually exclusive actions and, permission and obligation to perform mutually exclusive actions, in other words, for instance: the obligation to close the gate or open the gate but not both, permission to check the passport details or obligation to check the luggage but not both, respectively (Fenech et al., 2009b).

3 Natural language and Translation

We are aiming at a system for requirement analysis . The main characteristic of system is that the user is able to communicate with it in natural language. The communication with the system can take place in two ways. First, to deal with the properties of requirements and logical reasoning which expressed in NL, the system should be able to translate them into logical formulas specified in $\mathcal{CL}$ syntax which is then, it is possible to perform analysis and inferences with CLAN (Fenech et al., 2009b). Second, in order to be able to get the results back from those analysis to the user, it must be possible to translate the $\mathcal{CL}$ representation of the requirements back into NL. This is depicted in following figures:

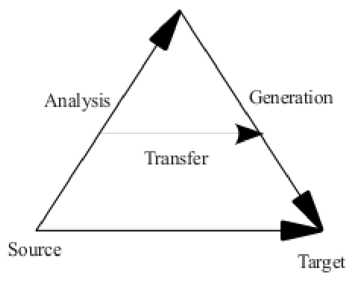

Figure 1: Vauquois Triangle for machine translation (Pace and Rosner, 2009)

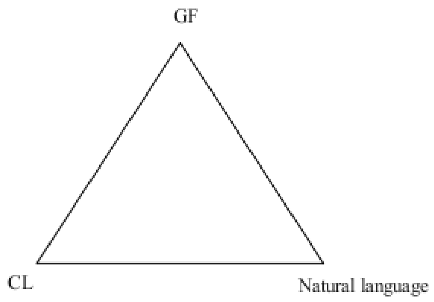

Figure 2: Natural Language triangle

It should be noted that in a classic transfer-based system there are generally three phases involved in the translation process which are (i) analysis (ii) transfer and (iii) generation (see figure 1). In this classical system the transfer phase "which applies transfer rules to abstract source sentence representation to yield target-language representations" is where the actual translation occur (Pace and Rosner, 2009) . The transfer rule in our case is what GF provides as an abstract and concrete syntax part where as depicted in the figure 1 our source language should be confined to the rules in abstract syntax once encodified and from there the concrete syntax yield target-language representation which is NL. What we consider in this paper as an analysis phase is quite different from what the classical approach defines. In our system this phase is performed by CLAN (Fenech et al., 2009b), whose the result will be fed into the system for further translation. The right hand vertex of the triangle represent the generation phase where a piece of text which fulfills the task will be generated from concrete syntax in a form of restricted natural language.

1 Restricted Natural Language

In our prototype, specific English words and sentences are defined in order to be able to represent $\mathcal{CL}$ syntaxes in natural language. The following table provide a complete picture of each $\mathcal{CL}$ syntax and its correspondence restricted English in our prototype:

Description	$\mathcal{CL}$	**English**
Obligation	$O(\alpha)$	It is mandatory to α
Permission	$P(\alpha)$	It is permitted to α
Prohibition	$F(\alpha)$	It is prohibited to α
CTD	$O_C(\alpha)$	It is mandatory to α if not α then any clause
CTP	$F_C(\alpha)$	It is prohibited to α if α then any clause
Test Operator	$[\beta]C$	If β then any clause
Conjunction	$C \wedge C$	any clause and any clause
XOR	$C - C$	any clause or any clause
Concurrency	$\alpha\ \&amp;\ \beta$	α and β
Sequence	$\alpha.\beta$	α then β
Choice	$\alpha + \beta$	α or β
Kleene star	$*\alpha$	As long as|Always|After α
Not	$!\alpha$	not α
Continuancy	$[1*](C \wedge C)$	As long as|Always|After any clause and any clause

Table 1: Restricted Natural Language

Suppose that we have part of original contract or requirements in NL and want to find out its correspondence $\mathcal{CL}$ formula by using the program described in this paper. The important point to consider is that, all of the sentences and clauses in the original contract or requirement at first step must be translated manually to English representation defined in table 1. This implies that without translating the original contract into above representation, the program would not be able to give its correspondence $\mathcal{CL}$ formula. Now let's look at the example below which is taken from a contract between Internet provider and a client (Pace et al., 2007):

> “ *The Provider is obliged to provide the Internet Sevices as stipulated in this Agreement* ”

To be able to manually translate this, the original concept of the sentence must be remembered in mind, and not the exact same wordings. Things such as traditional grammars like subject of sentence should be discarded and the only parts such as verbs and the concept of the sentence whether it expresses obligation or etc. must be preserved. Thus, in case of above example the sentence implies obligation of performing an action. By referring to table 1, the sentence can be expressed as : [Restricted NL] It is mandatory to provide the Internet Sevices as stipulated in this Agreement. In the end, to be able to feed this into the prototype, "provide the Internet Sevices as stipulated in this Agreement" must be added as an action (verb) to vocabulary part of the system (Action) modules.

4 Implementation

In this section we provide an implementation of GF grammars for $\mathcal{CL}$ syntaxes and natural language which in our case is English (see table 1). At first, we define categories and functions for handling different $\mathcal{CL}$ clauses and actions in abstract syntax part and then we move on to concrete syntax part where the representation of these in natural language will be expressed.

1 Abstract syntax

The abstract syntax module as a central structure contains the basis for representation and formalization of $\mathcal{CL}$ syntaxes. On the other hand, the linearization of the following functions and structures into $\mathcal{CL}$ symbolic syntaxes and natural language is simply done by the use of two concrete syntax modules where, with the first one, it would be possible to write the exact $\mathcal{CL}$ syntaxes, and with the latter, it would be possible to express them in the restricted natural language. Now let's begin by defining categories:

```
-- Abstract module(Cl.gf module)

cat Act  ;KleeneStarAct;KleeneCompAct;Clause;Clauses;ClauseP;↩
     ClauseO;ClauseF;And;Or;Dot;CompAct;Star;Not;
```

```
-- Concrete module(ClEng.gf module)

lincat
 Act,KleeneStarAct,KleeneCompAct,Clause,ClauseO,ClauseP,↩
      ClauseF,KleeneStarAct,KleeneCompAct,Clauses,And,Or,Dot↩
      ,Act,Star,Not,CompAct = {s : Str};
```

```
-- Concrete module(ClSym.gf module)

lincat Act,KleeneStarAct,KleeneCompAct,Clause,ClauseO,ClauseP↩
     ,ClauseF,KleeneStarAct,KleeneCompAct,Clauses,And,Or,Dot↩
     ,Act,Star,Not,CompAct = {s : Str};
```

Here we define category (type) such as `Clause` and `Act` (types of meanings) since in $\mathcal{CL}$, all the expressions are represented by certain clauses and different basic, compound and kleene star actions. Further we defined linearization type definitions to state that for instance `Clause` and `Act` are records with a string `s`. In following sections we will go through a detail explanation of the $\mathcal{CL}$ syntax in a form of function and linearization structure step by step to clearly show the process of converting these into restricted natural language and vice versa.

Obligation, Permission and Prohibition

As explained in the background section, obligation, permission and prohibition are three main clauses of $\mathcal{CL}$ which have the same structure in Cl module.

```
-- Abstract module(Cl.gf module)

fun
   Co : ClauseO -> Act -> Clause;
   Cp : ClauseP -> Act -> Clause;
   Cf : ClauseF -> Act -> Clause;

   Clo : ClauseO -> CompAct -> Clause;
   Clp : ClauseP -> CompAct -> Clause;
   Clf : ClauseF -> CompAct -> Clause;

   Obl : ClauseO -> CompAct -> Clauses;
   Per : ClauseP -> CompAct -> Clauses;
   Pro : ClauseF -> CompAct -> Clauses;

   Obligation : ClauseO -> Act -> Clauses;
   Permission : ClauseP -> Act -> Clauses;
   Prohibition : ClauseF -> Act -> Clauses;
```

All the functions above such as `Co`,`Cp` and `Cf` are used to represent `Obligation` , `Permission` and `Prohibition` respectively. Let's for the ease of explanation call the first three clauses, the first group, the second three clauses, the second group and so on. There are some differences among these four groups which worth mentioning them here. As it is clearly shown `Act` and `CompAct` are two different argument types used among the four groups which represent basic and compound actions respectively. This in effect, will provide the possibility to be able to express `Obligation`,`Permission` and `Prohibition` clauses with both basic and compound actions (Actions will be explained later on in this section). `ClauseO`,`ClauseP` and `ClauseF` are specifically designed to express obligation, permission and prohibition statements that together with the action (verb), form the actual clauses.

The other difference lies among these four groups and all the others that will be shown further on, is that generally a `Clause` itself can consist of different structures and thus in here a `Clause` can be either constructed from `ClauseO`,`ClauseP` and `ClauseF` together with basic or compound actions. However, it should be noted that they are defined here in this way to just facilitate the conjunction of clauses and exclusive or operation between clauses but not the direct linearization and parsing of each structure. For the latter purpose we specified `Clauses` as a start category for parsing and linearization so that each structure can be linearized and parsed directly. The linearization of these in concrete syntaxes are as follows:

```
-- Concrete module(ClEng.gf module)

lin
   Co clo acti = {s = clo.s ++ acti.s};
   Cp clp acti = {s = clp.s ++ acti.s};
   Cf clf acti = {s = clf.s ++ acti.s};

   Clo clo compact  = {s = clo.s ++ compact.s};
   Clp clp compact  = {s = clp.s ++ compact.s};
   Clf clf compact  = {s = clf.s ++ compact.s};

   Obl clo compact  = {s = clo.s ++ compact.s};
   Per clp compact  = {s = clp.s ++ compact.s};
   Pro clf compact  = {s = clf.s ++ compact.s};

   Obligation clo acti = {s = clo.s ++ acti.s};
   Permission clp acti = {s = clp.s ++ acti.s};
   Prohibition clf acti = {s = clf.s ++ acti.s};
```

Basically, `Obligation`, `Permission` and `Prohibition` clauses are expressed in natural language using restricted words "It is mandatory to" (`clo.s`), "It is permitted to" (`clp.s`) and "It is prohibited to" (`clf.s`) as terminals (quoted words in GF are called terminals (Ranta, A., 2009)) so that together with actions they formulate the clauses in NL.

As explained before in this paper, we provide another concrete syntax module called ClSym.gf to provide a possibility for the user to be able to write specific $\mathcal{CL}$ syntax such as operators, parentheses, brackets and etc to the program. This possibility is also vice versa which means if writing any specific clause in NL, the program with the help of this module would be able to provide $\mathcal{CL}$ formulas confine to what was provided in NL.

```
 -- Concrete module(ClSym.gf module)

lin
   Co clo acti = {s = clo.s ++ "(" ++ acti.s ++ ")"};
   Cp clp acti = {s = clp.s ++ "(" ++ acti.s ++ ")"};
   Cf clf acti = {s = clf.s ++ "(" ++ acti.s ++ ")"};

   Clo clo compact  = {s = clo.s ++ "(" ++ compact.s ++ ")"};
   Clp clp compact  = {s = clp.s ++ "(" ++ compact.s ++ ")"};
   Clf clf compact  = {s = clf.s ++ "(" ++ compact.s ++ ")"};

   Obl clo compact  = {s = clo.s ++ "(" ++ compact.s ++ ")"};
   Per clp compact  = {s = clp.s ++ "(" ++ compact.s ++ ")"};
   Pro clf compact  = {s = clf.s ++ "(" ++ compact.s ++ ")"};

   Obligation clo acti = {s = clo.s ++ "(" ++ acti.s ++ ")"};
   Permission clp acti = {s = clp.s ++ "(" ++ acti.s ++ ")"};
   Prohibition clf acti = {s = clf.s ++ "(" ++ acti.s ++ ")"↩
        };
```

The structure in above module is quite the same as ClEng.gf module with the only difference that `clo.s`, `clp.s` and `clf.s` each represent specific characters such as "`O`", "`P`" and "`F`" instead of words or sentences.

```
 -- Concrete module(ClSym.gf module)

lin
  O = {s = "O"};
  P = {s = "P"};
  F = {s = "F"};
```

It is illustrated that in linearization each function correspond to specific word.

CTDs and CTPs

Contrary-to-Duties (CTDs) and *Contrary-to-Prohibition* (CTPs) as both relate to obligation and prohibition clauses respectively could be expressed as following functions:

```
 -- Abstract module(Cl.gf module)

fun
   CTD,CTP  : Act -> Clause -> Clauses;

   CTDc,CTPc :  CompAct -> Clause -> Clauses;

   CTDbc,CTPbc  : Act -> Clause -> Clause;

   CTDcc,CTPcc  : CompAct -> Clause -> Clause;
```

From now on the same differences described in previous section will be applied to all other formalization and structures in following sections as well as this one. The reason for formulating the above structure is the fact that, CTD and CTP clauses are basically the obligation and prohibition of specific basic or compound actions which in case of violation, the reparation which can be itself another clause must be considered. For the sake of former and latter reason, it is needed to specify `Action` (compound or basic) and `Clause` as their argument types which together they build a clause representing the whole CTD or CTP. To avoid confusion different names has been used to declare function names.

```
 -- Concrete module(ClEng.gf module)

lin
    CTD acti clause  = {s = "It is mandatory to" ++ acti.s ++↩
         "if not" ++ acti.s ++ "then" ++ clause.s};
    CTP acti clause  = {s = "It is prohibited to" ++ acti.s ↩
        ++ "if" ++ acti.s ++ "then" ++ clause.s};

    CTDc compact clause = {s = "It is mandatory to" ++ ↩
         compact.s ++ "if not" ++ compact.s ++ "then" ++ ↩
         clause.s};
    CTPc compact clause = {s = "It is prohibited to" ++ ↩
         compact.s ++ "if" ++ compact.s ++ "then" ++ clause.↩
         s};

    CTDbc acti clause = {s = "It is mandatory to" ++ acti.s ↩
         ++ "if not" ++ acti.s ++ "then" ++ clause.s};
    CTPbc acti clause = {s = "It is prohibited to" ++ acti.s ↩
         ++ "if" ++ acti.s ++ "then" ++ clause.s};

    CTDcc compact clause = {s = "It is mandatory to" ++ ↩
         compact.s ++ "if not" ++ compact.s ++ "then" ++ ↩
         clause.s};
    CTPcc compact clause = {s = "It is prohibited to" ++ ↩
         compact.s ++ "if" ++ compact.s ++ "then" ++ clause.↩
         s};
```

In the linearization of CTD and CTP we have to notice that in CTD, one is obliged to perform specific action if violated then the reparation in a form of clause should be considered. This is also the same for CTP with the only difference that one is prohibited from doing specific action. Now, let's look at the structure in which the logical syntaxes of above is possible to express:

```
 -- Concrete module(ClSym.gf module)

lin
   CTD acti clause = {s = "O" ++ "(" ++ acti.s ++ ")" ++ "_" ↩
       ++ clause.s};
   CTP acti clause = {s = "F" ++ "(" ++ acti.s ++ ")" ++ "_" ↩
       ++ clause.s};

   CTDc compact clause = {s = "O" ++ "(" ++ compact.s ++ ")" ↩
       ++ "_" ++ clause.s};
   CTPc compact clause = {s = "F" ++ "(" ++ compact.s ++ ")" ↩
       ++ "_" ++ clause.s};

   CTDbc acti clause = {s = "O" ++ "(" ++ acti.s ++ ")" ++ "_↩
       " ++ clause.s} ;
   CTPbc acti clause = {s = "F" ++ "(" ++ acti.s ++ ")" ++ "_↩
       " ++ clause.s} ;

   CTDcc compact clause = {s = "O" ++ "(" ++ compact.s ++ ")"↩
        ++ "_" ++ clause.s};
   CTPcc compact clause = {s = "F" ++ "(" ++ compact.s ++ ")"↩
        ++ "_" ++ clause.s};
```

The only important thing to notice here is "`_`" character as also defined in (Fenech et al., 2009b) to express the reparation, meaning that the clause after this symbol is the reparation clause which has to be considered in case of a violation of the contract.

Conjunction of clauses ($C \wedge C$) and XOR operator

Generally, as it is illustrated in background section a $\mathcal{CL}$ clause can be constructed from conjunction of two clauses. Two clauses might have exclusive or operator in between to express the choice between two clauses but not both. Now, in order to be able to express the conjunction and exclusive or between two clauses the below representation is specified in GF:

```
-- Abstract module(Cl.gf module)

fun
   Conjunction : Clause -> Clause -> Clauses;
   ConjAlways :  Star -> Clause -> Clause -> Clauses;

   Xor : ClauseO -> Act -> ClauseO -> Act -> Clauses;
   Xorp : ClauseP -> Act -> ClauseP -> Act -> Clauses;

   XorCompAct : ClauseO -> CompAct -> ClauseO -> CompAct -> ↩
        Clauses;
   XorpCompAct : ClauseP -> CompAct -> ClauseP -> CompAct -> ↩
        Clauses;
```

As it is clearly defined in the above representation the structure used for conjunction of clauses consists of two clauses which may be any kind of clause such as obligation, prohibition, and etc. In order to be able to express repetition of action(s) used in conjunction of two clauses, the function `ConjAlways` has been specified to enhance the possibility to define structure (`Star`) of specific words that show continuanity. In effect, there is no limit in defining the conjunction operator between clauses with or without continuanity. However, It should be noted that as discussed before, the $\oplus$ operator is allowed to be used between certain clauses which is only between obligation and permission clauses.

```
-- Concrete module(ClEng.gf module)

lin
    Conjunction clause1 clause2 = {s = clause1.s ++ "and" ++ ↩
        clause2.s};
    ConjAlways star clause1 clause2 = {s = star.s ++ clause1.↩
        s ++ "and" ++ clause2.s};

    Xor clo acti clo acti1  = {s = clo.s ++ acti.s ++ "or" ++↩
         clo.s ++ acti1.s};
    Xorp clp acti clp acti1  = {s = clp.s ++ acti.s ++ "or" ↩
        ++ clp.s ++ acti1.s};

    XorCompAct clo compact clo compact1  = {s = clo.s ++ ↩
        compact.s ++ "or" ++ clo.s ++ compact1.s};
    XorpCompAct clp compact clp compact1  = {s = clp.s ++ ↩
        compact.s ++ "or" ++ clp.s ++ compact1.s};
```

what happens here is that we simply take two clauses and use "`and`" and "`or`" as terminals to express conjunction and exclusive choice between clauses. `star.s` represents words like "Always" in restricted English. For the symbolic linearization we define as below:

```
-- Concrete module(ClSym.gf module)

lin
    Conjunction clause1 clause2 = {s = clause1.s ++ "^" ++ ↩
        clause2.s};
    ConjAlways star clause1 clause2 = {s = "[" ++ "1" ++ star↩
        .s ++ "]" ++ "(" ++ clause1.s ++ "^" ++ clause2.s ↩
        ++ ")"};

    Xor clo acti clo acti1 = {s = clo.s ++ "(" ++ acti.s ++ "↩
        )" ++ "-" ++ clo.s ++ "(" ++ acti1.s ++ ")"};
    Xorp clp acti clp acti1 = {s = clp.s ++ "(" ++ acti.s ++ ↩
        ")" ++ "-" ++ clp.s ++ "(" ++ acti1.s ++ ")"};

    XorCompAct clo compact clo compact1  = {s = clo.s ++ "(" ↩
        ++ compact.s ++ ")" ++ "-" ++ clo.s ++ "(" ++ ↩
        compact1.s ++ ")"};
    XorpCompAct clp compact clp compact1 = {s = clp.s ++ "(" ↩
        ++ compact.s ++ ")" ++ "-" ++ clp.s ++ "(" ++ ↩
        compact1.s ++ ")"};
```

The structure used in above to show continuanity (`[1*]`) in conjunction of clauses, corresponds to what was defined in $\mathcal{CL}$. In order to be able to analyze XOR operation and the conjunction operator in CLAN, the operator should be represented as a dash line ("`-`") between clauses and "$\wedge$" operator symbolizes the conjunction of clauses.

Test Operator:

The grammar we specified in this paper should cover the whole $\mathcal{CL}$ syntax. Among them, there is specific case where we defined its name here as the test operator where it expresses conditional obligations, permissions and prohibitions (Pace et al., 2007). As mentioned in background section the test operator can be interpreted as: if specific action performs then the clause after it must be executed. More over, it is important to consider that repetition in actions or words in natural language, specify continuanity of the action(s), are usually shown by (*), which is allowed to be use inside the brackets (dynamic logic style) (Fenech et al., 2009b). Regarding this, specific type of actions known as Kleene star (see next section for more information) act shall be used inside the brackets. These actions are similar to regular actions except when some kind of repetition has been defined in the clause.

```
-- abstract module(Cl.gf module)

fun
   TestOp  : KleeneStarAct -> Clause -> Clauses;
   TestOpc : KleeneCompAct -> Clause -> Clauses;

   TestOpbc  : KleeneStarAct -> Clause -> Clause;
   TestOpcc  : KleeneCompAct -> Clause -> Clause;
```

Ideally, the test operator has the same structure as almost any other clauses except the actions which can be either kleene star basic action or compound one.

```
-- concrete module(ClEng.gf module)

fun
  TestOp kleenestaract clause = {s = "If" ++ kleenestaract.s ↩
      ++ "then"++ clause.s};
  TestOpc kleenecompact clause = {s = "If" ++ kleenecompact.s↩
       ++ "then"++ clause.s} ;

  TestOpbc kleenestaract clause = {s = "If" ++ kleenestaract.↩
      s ++ "then"++ clause.s};
  TestOpcc kleenecompact clause = {s = "If" ++ kleenecompact.↩
      s ++ "then"++ clause.s};
```

Notice the use of "`If`" and "`then`" words in specifying test operation in our restricted natural language.

```
-- concrete module(ClSym.gf module)

fun
 TestOp kleenestaract clause  = {s = "[" ++ kleenestaract.s ↩
     ++ "]" ++ clause.s};
 TestOpc kleenecompact clause = {s = "[" ++ kleenecompact.s ↩
      ++ "]" ++ clause.s} ;

 TestOpbc kleenestaract clause = {s = "[" ++ kleenestaract.s ↩
      ++ "]" ++ clause.s};
 TestOpcc kleenecompact clause = {s = "[" ++ kleenecompact.s ↩
      ++ "]" ++ clause.s};
```

To construct the test operator we should use brackets and for formulating the clause we use the same technique as before.

Actions:

Defining basic, compound and Kleene star actions in GF is an easy task, however, it should be noted that since actions in our case are generally verbs that could be considered as a specific vocabulary part (domain lexicon) (Ranta, A., 2009), it is more efficient to use module extension. This in effect separates the grammar part (`Cl module`) from a more specific vocabulary part (`Action module`) (Ranta, A., 2009). In other words, the user will be provided with modular system or program which means more flexibility to modify the modules and thus more maintainable system. In our case the Action module extends Cl module which is the main grammar.

```
 --  Action module(abstract syntax)

fun
   Pay,Buy : Act;
   CloseCheckIn,CorrectDetail : KleeneStarAct;
```

```
 --  Cl module (abstract syntax)

fun
   CompActa  : Act -> And -> Act -> CompAct;
   CompActo  : Act -> Or -> Act -> CompAct;
   CompActd  : Act -> Dot -> Act -> CompAct;
   CompActNeg  : Not -> Act -> CompAct;
   CompActNegc: Not -> CompAct -> CompAct;
```

The actions specified in above representation can be either single actions (basic and Kleene star) like "`pay`" and "`buy`" or as it is specified in `Cl module`, constructed from two basic actions with different operators in between as discussed in background section. The compound form of Kleenestar action has exactly the similar structure and same properties as regular compound action but just with different naming convention to apply and show the distinguishes. The linearization of the functions specified in `Action module` (abstract syntax) is easy to specify:

```
-- ActionEng module (concrete syntax)

lin
   Pay = {s = "pay"};
   Buy = {s = "buy"};
   CloseCheckIn = {s = "closeTheCheckIn"};
   CorrectDetail = {s = "checkThatThePassportDetailMatch"};
```

The structure of compound actions shows how the operators' name has been used as an argument types to build the functions. However, what we need to focus in translation of compound actions into the natural language is to know how each of the operators should be interpreted in natural language. As a consequence we end up with the following concrete syntax where it is possible to express all the operators:

```
-- Cl module (abstract syntax)

fun
   CompActa : Act -> And -> Act -> CompAct;
   CompActo : Act -> Or -> Act -> CompAct;
   CompActd : Act -> Dot -> Act -> CompAct;
   CompActNeg : Not -> Act -> CompAct;
   CompActNegc: Not -> CompAct -> CompAct;
```

```
-- ClEng module (concrete syntax)

lin
   CompActa acti and acti1 = {s = acti.s ++ and.s ++ acti1.s↩
        };
   CompActo acti or acti1 = {s = acti.s ++ or.s ++ acti1.s};
   CompActd acti dot acti1 = {s = acti.s ++ dot.s ++ acti1.s↩
        };
   CompActNeg not acti = {s = not.s ++ acti.s};
   CompActNegc not compact = {s = not.s ++ compact.s};
```

Thus, it is possible to express two actions happen concurrently (`and.s`), sequentially (`dot.s`), a choice (`or.s`) between two actions or even the negation (`not.s`) of the action. User defined operations such as the above fall under specific logical symbols which are defined in below:

```
-- ClSym module (concrete syntax)

lin
  CompActa acti and acti1 = {s = acti.s ++ and.s ++ acti1.s};
  CompActo acti or acti1 = {s = acti.s ++ or.s ++ acti1.s};
  CompActd acti dot acti1 = {s = acti.s ++ dot.s ++ acti1.s};
  CompActNeg not acti = {s = not.s ++ "(" ++ acti.s ++ ")"};
  CompActNegc not compact = {s = not.s ++ "(" ++ compact.s ++↩
        ")"};
```

In this manner, the representation of `and.s`, `or.s`, `dot.s`and `not.s` are confined to "`&`", "`+`", "`.`"and "`!`" logical operators respectively.

The Kleen star compound actions follow the similar structure as regular compound actions except two specific case:

```
-- Cl module (abstract syntax)

fun
  KleeneActs : KleeneStarAct -> Star -> KleeneCompAct;
  KleeneActcs: KleeneCompAct -> Star -> KleeneCompAct;
```

The above illustration show the possibility to express the continuanity of single and compound actions with the use of `Star` as an argument type.

The following excerpts show the correspondent concrete syntaxes which enable translation to natural and symbolic languages:

```
-- ClEng module (concrete syntax)

lin
  KleeneActs kleenestaract star = {s = star.s ++ ↩
        kleenestaract.s};
  KleeneActcs kleenecompact star = {s = star.s ++ ↩
        kleenecompact.s};
```

```
-- ClSym module (concrete syntax)

lin
  KleeneActs kleenestaract star = {s = kleenestaract.s ++ ↩
        star.s };
  KleeneActcs kleenecompact star = {s = "(" ++ kleenecompact.↩
        s ++ ")" ++ star.s };
```

The usage of `star.s` in ClEng module represents words such as "always", "as long as" in natural language and in ClSym module represents ”*“ symbol.

5 Case Study

In this section we provide part of a case study as an example in order to illustrate the process of conflict analysis by using the techniques mentioned earlier in this paper together with the help of CLAN. We start by showing an excerpt contract written in NL which is in this case, English and then representing this English contract, in a more restricted language (restricted English) as the one specified in this paper. The reason for the latter is to be able to feed this restricted contract written in English into the program explained in the former section to obtain $\mathcal{CL}$ logical formulas corresponding to what was provided as a restricted English contract. As a consequence we would be able to run this logical formulas through CLAN tool in order to observe whether a conflict exists in the requirements or not. Upon observing the conflict it would be possible to remove it from the $\mathcal{CL}$ formulas and thus, generate conflict free requirements represented in the restricted language. The contract is between an airline company and a company taking care of the ground crew and the case study focuses specifically on part of a contract related to check-in process (Fenech et al., 2009b). The extracted contract clauses expressed in restricted English and $\mathcal{CL}$, are given below (for complete English contract refer to appendix A):

1. The ground crew is obliged to open the check-in desk and request the passenger manifest two hours before the flight leaves.

   [Restricted English]: If two hours before the flight leaves then It is mandatory to open the check in desk and request the passenger manifest if not open the check in desk and request the passenger manifest then It is mandatory to issue a fine.

   [Program output]: [two hours before the flight leaves]O(open the check in desk &amp; request the passenger manifest) _ O(issue a fine)

2. After the check-in desk is opened the check-in crew is obliged to initiate the check-in process with any customer present by checking that the passport details match what is written on the ticket and that the luggage is within the weight limits. Then they are obliged to issue the boarding pass.

   [Restricted English]: If open the check in then It is mandatory to check that the passport details match and check that luggage is within the weight limits and If check that the passport detail match and check that luggage is within the weight limit then It is mandatory to issue the boarding pass if not issue the boarding pass then It is mandatory to issue a fine.

   [Program output]: [1*][open the check in]O(check that the passport details match &amp; check that luggage is within the weight limits) $\wedge$ [check that the passport detail match &amp; check that luggage is within the weight limit]O(issue the boarding pass) _ O(issue a fine)

3. The ground crew is obliged to close the check-in desk 20 minutes before the flight is due to leave and not before.

   [Restricted English]: If twenty minutes before the flight is due to leave and not before then It is mandatory to close the check in desk if not close the check in desk then It is mandatory to issue a fine and If not twenty minutes before the flight is due to leave and not before then It is prohibited to close the check in desk if close the check in desk then It is mandatory to issue a fine.

   [Program output]: [twenty minutes before the flight is due to leave and not before]O(close the check in desk) _ O(issue a fine) $\wedge$ [!(twenty minutes before the flight is due to leave and not before)]F(close the check in desk) _ O(issue a fine)

4. If any of the above obligations and prohibitions are violated a fine is to be paid.

   [Restricted English]: As long as If close the check in then It is prohibited to open the check in desk if open the check in desk then It is mandatory to issue a fine and It is prohibited to issue the boarding pass if issue the boarding pass then It is mandatory to issue a fine.

   [Program output]: [1*][close the check in]F(open the check in desk) _ O(issue a fine) $\wedge$ F(issue the boarding pass) _ O(issue a fine)

   As it is shown in the example above, once we run the logical formulas obtained from the program described in this paper through the CLAN, we would be able to observe the existence of any conflicts in the requirements however due to performance issue of the CLAN we only examine the clauses that contain conflicts in this case study. With the help of the CLAN, it would be possible to obtain instantaneous results showing any conflicts such as concurrent obligation and prohibition to perform specific action like *issue the boarding pass* that occur in the second and forth clause. If we look at second and forth clause, we will see that in the second clause one is obliged to *issue the boarding pass* if the client has correct information and as long as the check in desk opens. furthermore, in the forth clause one is forbidden to *issue the boarding pass* if the check in desk closes. The tool can easily identify an inconsistency between these two clauses. The problem we are considering here is when the client provide right information but check-in desk is closed.

Once the inconsistency identified by the CLAN tool, the problem can be easily fixed, for instance in our case the second clause can be changed to *It is mandatory for the ground crew to issue the boarding pass as long as the check in desk is not closed* (Fenech et al., 2009b).

6 Results

The primary obtained result is the actual encoding of $\mathcal{CL}$ syntax into GF abstract syntax and thus constructing the base structure in GF for expressing the logical clauses. This initial step is oriented towards identifying concrete syntaxes which facilitates the translation to natural languages and vice versa. At the same time, this is an essential step for covering the whole picture of automation and as a consequence could be

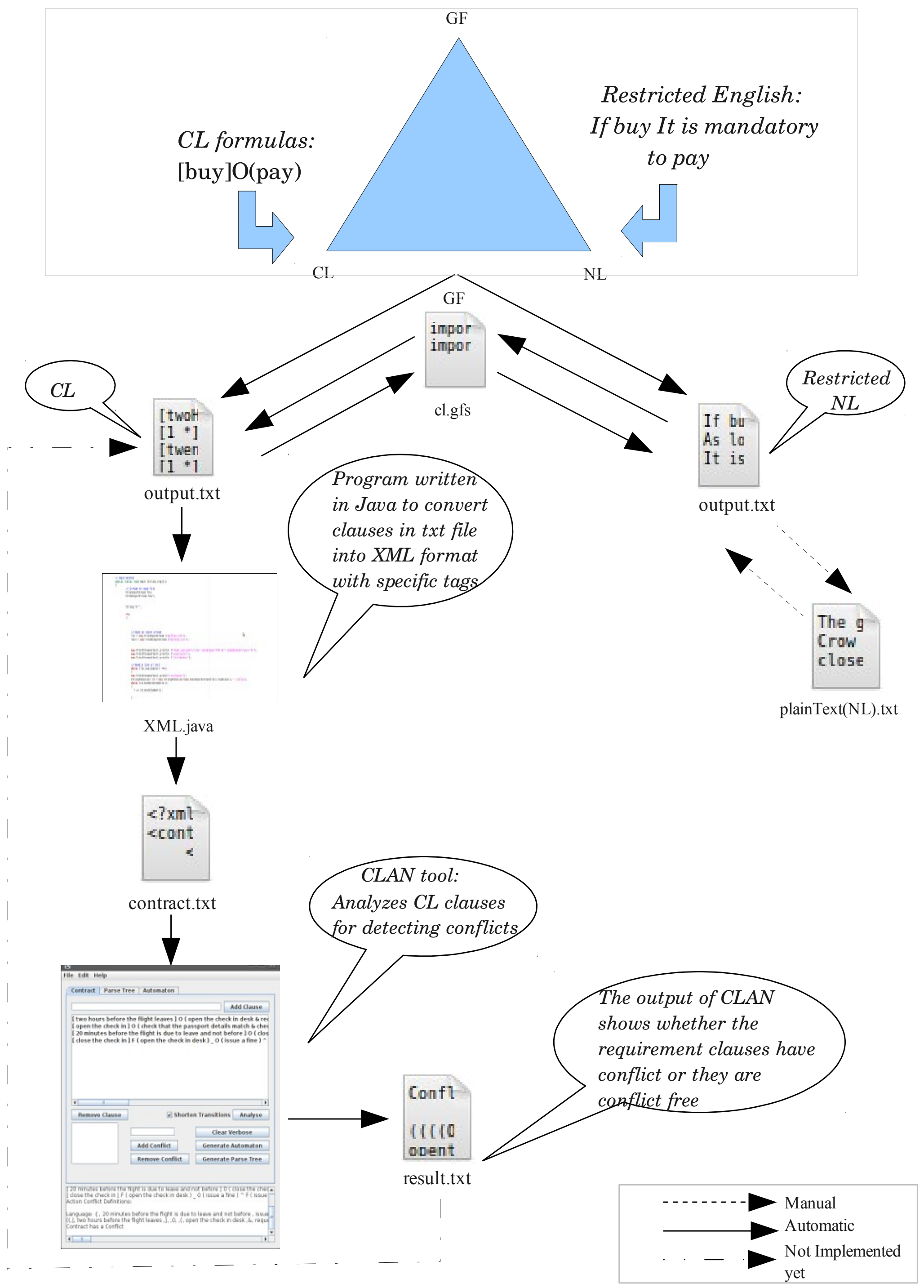

Figure 3: result

considered as our secondary result. We've merged different knowledge obtained from valuable sources together with our approach, to build a prototype that can be used for all sorts of specifications such as requirements, contracts, Internet-based negotiations and etc. which ensures the representation of these in formal language needed for analysis purpose and thus conflict detection. We used the data in the case study and have applied our approach to be able to present our results in a formal way. The following figure 3 is the illustration of the result of the whole process explained in this paper however, it should be noticed that, the research that has been conducted does not show the complete automation process as it needs more investigation. Discussions regarding the future contribution towards the full automation is described in Conclusion section.

The figure 3 is an illustration to show and prove the ability of the program explained in this paper that facilitates the process to identify contract inconsistencies and conflicts. For this purpose, the first step is to manually translate the original contracts written in NL (plain text in English)(see Appendix A) into Restricted NL (language needed as an input format to the prototype) (see Appendix B). After conducting the first step, we created a script file (see Appendix E) for the prototype which automatically import the libraries (also grammar: the modules written in GF) as well as specific commands which can linearize and parse the input (Restricted NL or CL) and thus provides the result in an output file like output.txt. However, it should be noticed that the path for using libraries should be set by the user. To be able to use the prototype, after installing GF, one must use the following command : gf - -<cl.gfs in GF editor which will provide the user with an output file in txt format. Once obtaining logical clauses (please refer to Case study for more information) (see Appendix C) as an output file, we feed in the output file into a program (XML.java) written in Java to obtain what is needed as input file in XML format for CLAN (see Appendix D). This can be done by compiling and running the program by using the ordinary commands (i) javac XML.java (ii) java XML. The file obtained from this program contains all the logical clauses correspond to what was specified as Restricted NL and in XML format, which is needed as an input for CLAN. As a final step CLAN tool has been able to analyze these clauses by using the command java -jar clan-gui-1.0.0-clan.jar contract.txt result.txt which provides the analysis result in ordinary text file showing whether the contract contains conflict or not. In the case of existence of any conflicts in the requirements, the CLAN will also provide a counter example to show the trace to the conflicts and the conflict clause itself (see Appendix F).

7 Related Work

There are no earlier efforts regarding the actual translation of $\mathcal{CL}$ to natural language however, researches have been conducted where GF was used in order to implement specific language. Hähnle, Johannisson and Ranta (Hahnle et al., 2002) talk about design principle of a tool which helps to authorize formal and informal software requirement specifications. "The tool is an attempt to bridge the gap between completely informal requirements specifications (as found in practice) and formal ones (as needed in formal methods)" (Hahnle et al., 2002). Some of the problems outlined in the paper like authoring well-formed formal specifications, maintenance, mapping different levels of formality and synchronization made them to come up with the solution to handle these problems. The solution outlined in the paper is an illustration of the possibility of connection between specification languages in different levels. Their focus as stated in the paper is on Object Constraint Language (OCL) and Natural Language (NL) as specification languages and their approach is based on Grammatical Framework. They managed to implement different concepts of OCL such as, Classes and Objects, Attributes, Operations and Queries in GF. Pace and Rosner (Pace and Rosner, 2009) presented an end-user system which is specifically design to process the domain of computer oriented contracts. The main abilities of the system as described in the paper are logical reasoning about the properties of contracts which relates to internal consistency of a contract and about the status of actions whether they are prohibited or permitted. They use (controlled natural language) CNL to specify contracts and they also define an appropriate logic much the same as CL to analyze CNL contracts.

8 Conclusions

Generally, contracts or requirements are required to be consistent, to be able to enable safe and dependable contract adoption (Fenech et al., 2009b) among organizations. In this paper we have presented an encoding of the contract language $\mathcal{CL}$ into GF, and back. We have also implemented a program which can be considered as a prototype that allows the translation back and forth between $\mathcal{CL}$ and NL, and applied this to a case study in which we obtained valuable results (for more information refer to result section). This research was specifically conducted to detect conflicts in order to reduce inaccuracies in requirements by using a formal language, $\mathcal{CL}$ and its implementation in GF. Currently, this prototype tool with the help of CLAN enhances the process of identifying conflict. However, the functionality of translating the conflict clauses and counter examples obtained from CLAN, in case of existence of conflict into restricted natural language is part of our future work. Furthermore, other functionalities such as Passage Retrieval (Buscaldi et al., 2009) which enables the NL to map to its correspondent Restricted NL are to be investigated and added to the program.

9 Acknowledgments

We would like to thank Aarne Ranta and Stephen Fenech for all their supports and guidance with providing ideas and feedback during this research. Special thanks to Gerardo Schneider for all the ideas and supports that makes this research possible to be presented to you.

References

(2009). Requirements contract Wikipedia. http://en.wikipedia.org/wiki/Requirements_contract. 2

(2010). Saab in brief. http://www.saabgroup.com/en/About-Saab/Company-profile/Saab-in-brief/. 2

Aagedal, J. (2001). *Quality of service support in development of distributed systems*. PhD thesis, Citeseer. 3

Berling, T. (2010). Personal Communication Saab . 2

Beugnard, A., Jézéquel, J., Plouzeau, N., and Watkins, D. (1999). Making components contract aware. *Computer*, 32(7):38–45. 3

Buscaldi, D., Rosso, P., Gómez-Soriano, J., and Sanchis, E. (2009). Answering Questions with an n-gram based Passage Retrieval Engine. *Journal of Intelligent Information Systems*, pages 1–22. 12

Creswell, J. (2008). *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage Pubns. 3

Daskalopulu, A. (2000). Model checking contractual protocols. In *Legal knowledge and information systems: JURIX 2000: the thirteenth annual conference*, page 35. Ios Pr Inc. 3

Daskalopulu, A. and Maibaum, T. (2001). Towards electronic contract performance. In *dexa*, page 0771. Published by the IEEE Computer Society. 3

Davulcu, H., Kifer, M., and Ramakrishnan, I. (2004). CTR-S: a logic for specifying contracts in semantic web services. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 144–153. ACM. 3

Fenech, S., Pace, G., and Schneider, G. (2009a). Automatic Conflict Detection on Contracts. *Theoretical Aspects of Computing-ICTAC 2009*, pages 200–214. 14

Fenech, S., Pace, G., and Schneider, G. (2009b). CLAN: A Tool for Contract Analysis and Conflict Discovery. In *Automated Technology for Verification and Analysis: 7th International Symposium, Atva 2009, Macao, China, October 14-16, 2009, Proceedings*, page 90. Springer. 3, 5, 7, 8, 10, 12

Governatori, G. (2005). Representing business contracts in RuleML. *International Journal of Cooperative Information Systems*, 14(2-3):181–216. 3

Governatori, G. and Rotolo, A. (2006). Logic of violations: A Gentzen system for reasoning with contrary-to-duty obligations. *Australasian Journal of Logic*, 4:193–215. 3

Hahnle, R., Johannisson, K., and Ranta, A. (2002). An authoring tool for informal and formal requirements specifications. *Lecture notes in computer science*, pages 233–248. 2, 4, 12

Hevner, A., March, S., Park, J., and Ram, S. (2004). Design science in information systems research. *Management information systems quarterly*, 28(1):75–106. 2, 3

Molina-Jimenez, C., Shrivastava, S., Solaiman, E., and Warne, J. (2004). Run-time monitoring and enforcement of electronic contracts. *Electronic Commerce Research and Applications*, 3(2):108–125. 3

Osterwalder, A. (2004). The Business Model Ontology-a proposition in a design science approach. *Academic Dissertation, Universite de Lausanne, Ecole des Hautes Etudes Commerciales*. 2

Pace, G., Prisacariu, C., and Schneider, G. (2007). Model checking contracts-a case study. *Lecture Notes in Computer Science*, 4762:82. 4, 5, 8

Pace, G. and Rosner, M. (2009). A Controlled Language for the Specification of Contracts. 5, 12

Paschke, A., Dietrich, J., and Kuhla, K. (2005). A logic based sla management framework. In *4th Semantic Web Conference (ISWC 2005)*. Citeseer. 3

Prisacariu, C. and Schneider, G. (2007). Towards a formal definition of electronic contracts. Technical report, Technical Report 348, Department of Informatics, University of Oslo, Oslo, Norway. 3

Prisacariu, C. and Schneider, G. (2009). CL: An Action-based Logic for Reasoning about Contracts. *WOLLIC'09*, 5514:335–349. 2, 3

Ranta, A. (2009). Grammatical framework : A programming language for multilingual grammars and their applications. 1, 2, 4, 7, 9

Robinson, W. N. and Pawlowski, S. D. (1999). Managing requirements inconsistency with development goal monitors. *IEEE Trans. Software Eng.*, 25(6):816–835. 1

Song, I. and Governatori, G. (2004). Nested rules in defeasible logic. *Rules and Rule Markup Languages for the Semantic Web*, pages 204–208. 3

Van Aken, J. (2005). Management research as a design science: articulating the research products of mode 2 knowledge production in management. *British Journal of Management*, 16(1):19–36. 2, 3

A Plain text of Case study

Below is original contract clauses in plain English (Fenech et al., 2009a):

1. *The ground crew is obliged to open the check-in desk and request the passenger manifest two hours before the flight leaves.*
2. *The airline is obliged to reply to the passenger manifest request made by the ground crew when opening the desk with the passenger manifest.*
3. *After the check-in desk is opened the check-in crew is obliged to initiate the check-in process with any customer present by checking that the passport details match what is written on the ticket and that the luggage is within the weight limits. Then they are obliged to issue the boarding pass.*
4. *If the luggage weighs more than the limit, the crew is obliged to collect payment for the extra weight and issue the boarding pass.*
5. *The ground crew is prohibited from issuing any boarding cards without inspecting that the details are correct beforehand.*
6. *The ground crew is prohibited from issuing any boarding cards before opening the check-in desk.*
7. *The ground crew is obliged to close the check-in desk 20 minutes before the flight is due to leave and not before.*
8. *After closing check-in, the crew must send the luggage information to the airline.*
9. *Once the check-in desk is closed, the ground crew is prohibited from issuing any boarding pass or from reopening the check-in desk.*
10. *If any of the above obligations and prohibitions are violated a fine is to be paid.*

B Restricted NL of Case study

Below is the representation of extracted contracts in Restricted language:

1. If twoHoursBeforeTheFlightLeaves then It is mandatory to openTheCheckInDesk and requestThePassengerManifest if not openTheCheckInDesk and requestThePassengerManifest then It is mandatory to issueAFine.
2. After If openTheCheckIn then It is mandatory to checkThatThePassportDetailsMatch and checkThatLuggageIsWithinTheWeightLimits and If checkThatThePassportDetailMatch and checkThatLuggageIsWithinTheWeightLimit then It is mandatory to issueTheBoardingPass if not issueTheBoardingPass then It is mandatory to issueAFine.
3. If twentyMinutesBeforeTheFlightIsDueToLeaveAndNotBefore then It is mandatory to closeTheCheckInDesk if not closeTheCheckInDesk then It is mandatory to issueAFine and If not twentyMinutesBeforeTheFlightIsDueToLeaveAndNotBefore then It is prohibited to closeTheCheckInDesk if closeTheCheckInDesk then It is mandatory to issueAFine.
4. As long as If closeTheCheckIn then It is prohibited to openTheCheckInDesk if openTheCheckInDesk then It is mandatory to issueAFine and It is prohibited to issueTheBoardingPass if issueTheBoardingPass then It is mandatory to issueAFine

C $\mathcal{CL}$ of Case study

below is the logical clauses obtained from the prototype:

1. [two hours before the flight leaves] O(open the check in desk & request the passenger manifest) _ O(issue a fine)
2. [1*]([open the check in] O(check that the passport details match & check that luggage is within the weight limits) ∧ [check that the passport detail match & check that luggage is within the weight limit]O(issue the boarding pass) _ O(issue a fine))
3. [20 minutes before the flight is due to leave and not before]O(close the check in desk) _ O(issue a fine) ∧ [!(20 minutes before the flight is due to leave and not before)]F(close the check in desk) _ O(issue a fine)
4. [1*]([close the check in]F(open the check in desk) _ O(issue a fine) ∧ F(issue the boarding pass) _ O(issue a fine))

D XML output

The following is the logical clauses in XML format: <?XML version="1.0" encoding="UTF-8" standalone="yes"?>
<contract>
<clauses>
<clause>[1∗]([open The Check In] O(check That The Passport Details Match &amp; check That Luggage Is Within The Weight Limits) ∧ [check That The Passport Detail Match &amp; check That Luggage Is Within The Weight Limit] O(issue The Boarding Pass)_O(issue A Fine))</clause>
<clause>[1∗]([close The Check In] F(open The Check In Desk)_O(issue A Fine) ∧ F(issue The Boarding Pass)_O(issue A Fine))</clause>
</clauses>
<concurrentActions/>
</contract>

E Script file: cl.gfs

```
import /home/shayan/test/ActionEng.gf
import /home/shayan/test/ActionSym.gf


parse −lang=ActionEng "If twoHoursBeforeTheFlightLeaves then It is mandatory to openTheCheckInDesk ↩
    and requestThePassengerManifest if not openTheCheckInDesk and requestThePassengerManifest then ↩
    It is mandatory to issueAFine" | linearize −lang=ActionSym | put_string −unlexcode | write_file ↩
    −append −file=output.txt

parse −lang=ActionEng "After If openTheCheckIn then It is mandatory to ↩
    checkThatThePassportDetailsMatch and checkThatLuggageIsWithinTheWeightLimits and If ↩
    checkThatThePassportDetailMatch and checkThatLuggageIsWithinTheWeightLimit then It is mandatory ↩
    to issueTheBoardingPass if not issueTheBoardingPass then It is mandatory to issueAFine" | ↩
    linearize −lang=ActionSym | put_string −unlexcode | write_file −append −file=output.txt

parse −lang=ActionEng "If twentyMinutesBeforeTheFlightIsDueToLeaveAndNotBefore then It is mandatory ↩
    to closeTheCheckInDesk if not closeTheCheckInDesk then It is mandatory to issueAFine and If not ↩
    twentyMinutesBeforeTheFlightIsDueToLeaveAndNotBefore then It is prohibited to ↩
    closeTheCheckInDesk if closeTheCheckInDesk then It is mandatory to issueAFine" | linearize −lang↩
    =ActionSym | put_string −unlexcode | write_file −append −file=output.txt

parse −lang=ActionEng "As long as If closeTheCheckIn then It is prohibited to openTheCheckInDesk if ↩
    openTheCheckInDesk then It is mandatory to issueAFine and It is prohibited to ↩
    issueTheBoardingPass if issueTheBoardingPass then It is mandatory to issueAFine" | linearize −↩
    lang=ActionSym | put_string −unlexcode | write_file −append −file=output.txt

put_string −lexcode "[twoHoursBeforeTheFlightLeaves]O (openTheCheckInDesk &amp; ↩
    requestThePassengerManifest)_ O (issueAFine)" | parse −lang=ActionSym | linearize −lang=↩
    ActionEng

put_string −lexcode "[ 1 * ] ( [ openTheCheckIn ] O ( checkThatThePassportDetailsMatch &amp; ↩
    checkThatLuggageIsWithinTheWeightLimits ) ^ [ checkThatThePassportDetailMatch &amp; ↩
    checkThatLuggageIsWithinTheWeightLimit ] O ( issueTheBoardingPass ) _ O ( issueAFine ) )" | ↩
    parse −lang=ActionSym | linearize −lang=ActionEng

put_string −lexcode "[twentyMinutesBeforeTheFlightIsDueToLeaveAndNotBefore]O (closeTheCheckInDesk)_ O↩
     (issueAFine)^ [! (twentyMinutesBeforeTheFlightIsDueToLeaveAndNotBefore)]F (closeTheCheckInDesk)↩
    _ O (issueAFine)" | parse −lang=ActionSym |linearize −lang=ActionEng

put_string −lexcode "[ 1 * ] ( [ closeTheCheckIn ] F ( openTheCheckInDesk ) _ O ( issueAFine ) ^ F ( ↩
    issueTheBoardingPass ) _ O ( issueAFine ) ) " | parse −lang=ActionSym | linearize −lang=↩
    ActionEng |  write_file −append −file=output.txt
```

F Result of CLAN: result.txt

```
Conflict

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^([1]([(*1)]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))↩
    )^(((OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^([1]([(*1)]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))))))^((((F(openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn]((F(↩
    openTheCheckInDesk)_(OissueAFine)))^([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(↩
    OissueAFine))))))^((OissueAFine)^(((F(issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(↩
    issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass&closeTheCheckIn

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^([1]([(*1)]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))↩
    )^(((OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^([1]([(*1)]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))))))^(((OissueAFine)^(((F(openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn↩
    ]((F(openTheCheckInDesk)_(OissueAFine)))^([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(↩
    OissueAFine))))))))^(((F(issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(↩
    issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass&closeTheCheckIn,openTheCheckIn&checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueTheBoardingPass&issueAFine&closeTheCheckIn&↩
    openTheCheckInDesk

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^([1]([(*1)]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))))^(((OissueTheBoardingPass)_(OissueAFine))^(([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine)))^([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))^(((↩
    OissueAFine)^(((F(openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn]((F(openTheCheckInDesk)_↩
    (OissueAFine)))^([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine))))))))^(((F(↩
    issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass&closeTheCheckIn,checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueTheBoardingPass&issueAFine&closeTheCheckIn&↩
    openTheCheckInDesk

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^([1]([(*1)]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))↩
    )^(((OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^([1]([(*1)]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))))))^(((OissueAFine)^(([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))↩
    ^([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))))))^(((F(↩
    issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass&closeTheCheckIn,openTheCheckIn&checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueTheBoardingPass&issueAFine&openTheCheckInDesk

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^([1]([(*1)]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))))^(((OissueTheBoardingPass)_(OissueAFine))^(([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine)))^([1]([(*1)]([[(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))^(((↩
    OissueAFine)^(([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))^([1]([(*1)]([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))))))^(((F(issueTheBoardingPass)_(↩
    OissueAFine))^([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass&closeTheCheckIn,checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueTheBoardingPass&issueAFine&openTheCheckInDesk

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^([1]([(*1)]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))↩
    )^((OissueAFine)^(((OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^([1]([(*1)]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))))))))^((((F(openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn]((F(↩
    openTheCheckInDesk)_(OissueAFine)))^([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(↩
    OissueAFine)))))))^(((F(issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(issueTheBoardingPass↩
    )_(OissueAFine)))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass&closeTheCheckIn,openTheCheckIn&checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueAFine&closeTheCheckIn

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^([1]([(*1)]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))))^((OissueAFine)^(((OissueTheBoardingPass)_(↩
    OissueAFine))^(([(checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((↩
    OissueTheBoardingPass)_(OissueAFine)))^([1]([(*1)]([[(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))))))))^((((F(↩
    openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))↩
    ^([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))))))^(((F(↩
    issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass&closeTheCheckIn,checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueAFine&closeTheCheckIn

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^([1]([(*1)]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))↩
    )^((OissueAFine)^(((OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^([1]([(*1)]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))))))))^((([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))^([1]([(*1)]([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine))))))^(((F(issueTheBoardingPass)_(↩
    OissueAFine))^([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass&closeTheCheckIn,openTheCheckIn&checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueAFine

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^([1]([(*1)]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))))^((OissueAFine)^(((OissueTheBoardingPass)_(↩
    OissueAFine))^(([(checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((↩
    OissueTheBoardingPass)_(OissueAFine)))^([1]([(*1)]([[(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))))))))^((([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))^([1]([(*1)]([closeTheCheckIn]((F(↩
    openTheCheckInDesk)_(OissueAFine))))))^(((F(issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(↩
    issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass&closeTheCheckIn,checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueAFine

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^(([1]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))^([1]([1]([(*1)]([openTheCheckIn](O(↩
    checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))))^(((↩
    OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^(([1]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))^([1]([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))))))))^((((F(↩
    openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))↩
    ^(([1]([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine))))^([1]([1]([(*1)]([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))))))))^((OissueAFine)^(((F(↩
    issueTheBoardingPass)_(OissueAFine))^(([1]((F(issueTheBoardingPass)_(OissueAFine)))↩
    ^([1]([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine))))))))))
checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&issueTheBoardingPass&↩
    closeTheCheckIn

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^([1]([(*1)]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))↩
    )^(((OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^([1]([(*1)]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))))))^((((F(openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn]((F(↩
    openTheCheckInDesk)_(OissueAFine)))^([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(↩
    OissueAFine)))))))^(((F(issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(issueTheBoardingPass↩
    )_(OissueAFine)))))))
openTheCheckIn&issueTheBoardingPass&closeTheCheckIn,openTheCheckIn&checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueAFine&closeTheCheckIn

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^([1]([(*1)]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))))^(((OissueTheBoardingPass)_(OissueAFine))^(([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine)))^([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))^((((F(↩
    openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))↩
    ^([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))))))^(((F(↩
    issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&issueTheBoardingPass&closeTheCheckIn,checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueAFine&closeTheCheckIn

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^([1]([(*1)]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))↩
    )^(((OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^([1]([(*1)]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))))))^((([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))^([1]([(*1)]([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine))))))^(((F(issueTheBoardingPass)_(↩
    OissueAFine))^([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&issueTheBoardingPass&closeTheCheckIn,openTheCheckIn&checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueAFine

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^([1]([(*1)]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))))^(((OissueTheBoardingPass)_(OissueAFine))^(([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine)))^([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))^((([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))^([1]([(*1)]([closeTheCheckIn]((F(↩
    openTheCheckInDesk)_(OissueAFine))))))^(((F(issueTheBoardingPass)_(OissueAFine))^([1]([(*1)]((F(↩
    issueTheBoardingPass)_(OissueAFine)))))))
openTheCheckIn&issueTheBoardingPass&closeTheCheckIn,checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits&checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit&issueAFine

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^(([1]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))↩
    ^([1]([1]([(*1)]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits)))))))))^(((OissueTheBoardingPass)_(OissueAFine))^(([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine)))^(([1]([(checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)↩
    ]((OissueTheBoardingPass)_(OissueAFine))))^([1]([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))))^((((F(↩
    openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))↩
    ^(([1]([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine))))^([1]([1]([(*1)]([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))))))))^(((F(issueTheBoardingPass)_(↩
    OissueAFine))^(([1]((F(issueTheBoardingPass)_(OissueAFine)))^([1]([1]([(*1)]((F(↩
    issueTheBoardingPass)_(OissueAFine)))))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&closeTheCheckIn

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^(([1]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))^([1]([1]([(*1)]([openTheCheckIn](O(↩
    checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))))^(((↩
    OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^(([1]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))^([1]([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))))^((((F(↩
    openTheCheckInDesk)_(OissueAFine))^(([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))↩
    ^(([1]([closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine))))^([1]([1]([(*1)]([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))))))))^(((F(issueTheBoardingPass)_(↩
    OissueAFine))^(([1]((F(issueTheBoardingPass)_(OissueAFine)))^([1]([1]([(*1)]((F(↩
    issueTheBoardingPass)_(OissueAFine)))))))))
checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&closeTheCheckIn

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^(([1]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))↩
    ^([1]([1]([(*1)]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits)))))))))^(((OissueTheBoardingPass)_(OissueAFine))^(([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine)))^(([1]([(checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)↩
    ]((OissueTheBoardingPass)_(OissueAFine))))^([1]([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))))^((([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))^(([1]([closeTheCheckIn]((F(↩
    openTheCheckInDesk)_(OissueAFine))))^([1]([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(↩
    OissueAFine))))))))^((OissueAFine)^(((F(issueTheBoardingPass)_(OissueAFine))^(([1]((F(↩
    issueTheBoardingPass)_(OissueAFine)))^([1]([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))↩
    )))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&↩
    issueTheBoardingPass

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^(([1]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))^([1]([1]([(*1)]([openTheCheckIn](O(↩
    checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))))^(((↩
    OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^(([1]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))^([1]([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))))^((([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))^(([1]([closeTheCheckIn]((F(↩
    openTheCheckInDesk)_(OissueAFine))))^([1]([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(↩
    OissueAFine))))))))^((OissueAFine)^(((F(issueTheBoardingPass)_(OissueAFine))^(([1]((F(↩
    issueTheBoardingPass)_(OissueAFine)))^([1]([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))↩
    )))))
checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit&issueTheBoardingPass

((((O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))^(([openTheCheckIn](O↩
    (checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))^(([1]([↩
    openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))↩
    ^([1]([1]([(*1)]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits)))))))))^(((OissueTheBoardingPass)_(OissueAFine))^(([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine)))^(([1]([(checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)↩
    ]((OissueTheBoardingPass)_(OissueAFine))))^([1]([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))))^((([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))^(([1]([closeTheCheckIn]((F(↩
    openTheCheckInDesk)_(OissueAFine))))^([1]([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(↩
    OissueAFine))))))))^(((F(issueTheBoardingPass)_(OissueAFine))^(([1]((F(issueTheBoardingPass)_(↩
    OissueAFine)))^([1]([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))))))
openTheCheckIn&checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit

(((([openTheCheckIn](O(checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits)))↩
    ^(([1]([openTheCheckIn](O(checkThatThePassportDetailsMatch&↩
    checkThatLuggageIsWithinTheWeightLimits))))^([1]([1]([(*1)]([openTheCheckIn](O(↩
    checkThatThePassportDetailsMatch&checkThatLuggageIsWithinTheWeightLimits))))))))^(((↩
    OissueTheBoardingPass)_(OissueAFine))^(([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine)))^(([1]([(↩
    checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)↩
    _(OissueAFine))))^([1]([1]([(*1)]([(checkThatThePassportDetailMatch&↩
    checkThatLuggageIsWithinTheWeightLimit)]((OissueTheBoardingPass)_(OissueAFine))))))))))^((([↩
    closeTheCheckIn]((F(openTheCheckInDesk)_(OissueAFine)))^(([1]([closeTheCheckIn]((F(↩
    openTheCheckInDesk)_(OissueAFine))))^([1]([1]([(*1)]([closeTheCheckIn]((F(openTheCheckInDesk)_(↩
    OissueAFine))))))))^(((F(issueTheBoardingPass)_(OissueAFine))^(([1]((F(issueTheBoardingPass)_(↩
    OissueAFine)))^([1]([1]([(*1)]((F(issueTheBoardingPass)_(OissueAFine)))))))))
checkThatThePassportDetailMatch&checkThatLuggageIsWithinTheWeightLimit
```