From Contracts in Structured English to $\mathscr{CL}$ Specifications

Seyed Morteza Montazeri, Nivir Kanti Singha Roy
Department of Computer Science and Engineering
University of Gothenburg, Sweden
gusmonse@student.gu.se, guskanthni@student.gu.se

Gerardo Schneider
Department of Computer Science and Engineering
Chalmers | University of Gothenburg, Sweden
Department of Informatics
University of Oslo, Norway
gersch@chalmers.se

In this paper we present a framework to analyze conflicts of contracts written in structured English. A contract that has manually been rewritten in a structured English is automatically translated into a formal language using the Grammatical Framework (GF). In particular we use the contract language $\mathscr{CL}$ as a target formal language for this translation. In our framework $\mathscr{CL}$ specifications could then be input into the tool CLAN to detect the presence of conflicts (whether there are contradictory obligations, permissions, and prohibitions. We also use GF to get a version in (restricted) English of $\mathscr{CL}$ formulae. We discuss the implementation of such a framework.

1 Introduction

Natural language (e.g., English) descriptions and prescriptions abound on documents used in different phases of the software development process, including informal specifications, requirements, and contracts at different levels (methods/functions, objects, components, services, etc.). There is no doubt of the usefulness of having such descriptions and prescriptions in natural language (NL), as most of the intended users of the corresponding documents would not have problems to understand them. However, it is well known that NL is ambiguous and imprecise in many cases due to context sensitivity, underspecified terminology, or simply bad use of the language.

At the other extreme, we have formal methods with a myriad of different formal languages (logics) with complex syntax and semantics. Those languages are indeed extremely useful as they are precise, unambiguous, and in many cases tools are available as to provide the possibility of (semi-) automatic analysis. However, in many cases they require high expertise not only at the syntactic level in order to use the language to specify system properties or requirements, but also to interpret the results of the tools. For instance, though the use of model checkers has been advertised as a "push-button technology" not requiring user expertise, the reality is that one still needs to write the properties on a logic and interpret the counter-examples which are also usually given as a big formula representing the trace leading to the problematic case.

On an ideal world, software engineers (and the mortal non-technical users) would only need to deal with natural language descriptions, push a button and get a result telling them whether for instance the given contract (specifications, set of requirements)[1] is consistent and conflict-free. The current state-of-practice however is far from that ideal world. Though the state of the art on NL processing has advanced

[1] In the rest of the paper we will use the term "contract" to refer to contracts at different levels (including legal contracts), software specifications, requirements, etc.

E. Pimentel, V. Valero (Eds.): Workshop on Formal Languages and Analysis of Contract-Oriented Software 2011 (FLACOS'11)
EPTCS 68, 2011, pp. 55–69, doi:10.4204/EPTCS.68.6

$$
\begin{array}{rcl@{\qquad}rcl}
C & := & C_O \mid C_P \mid C_F \mid C \wedge C \mid [\beta]C \mid \top \mid \bot & C_F & := & F_C(\alpha) \\
C_O & := & O_C(\alpha) \mid C_O \oplus C_O & \alpha & := & 0 \mid 1 \mid a \mid \alpha \& \alpha \mid \alpha.\alpha \mid \alpha + \alpha \\
C_P & := & P(\alpha) \mid C_P \oplus C_P & \beta & := & 0 \mid 1 \mid a \mid \beta \& \beta \mid \beta.\beta \mid \beta + \beta \mid \beta^*
\end{array}
$$

Figure 1: $\mathscr{CL}$ syntax.

quite a lot in recent years, we still have to depend on the use of formal languages and techniques to analyze such contracts. A relatively new trend is then to restrict the use of NL in order to get something that "looks like a NL" but it has a better structure, and if possible avoid ambiguity. We call such constrained languages *restricted* or *controlled* NL.

Our aim in this paper is to advance towards finding a suitable solution to the problems and challenges just mentioned, in particular the possibility of writing contracts in NL, but being able to analyze them with tools, automating the process as much of possible. In particular, we show that it is possible to relate the formal language for contracts $\mathscr{CL}$ [10] and a restricted NL by using the Grammatical Framework (GF) [11]. In this way we are able to take contracts written in NL, manually obtain a restricted NL version of such contract, use GF to automatically obtain a $\mathscr{CL}$ formula that could then be analyzed using CLAN [3]. Note that the above process should be in both directions, as we might need the translation from $\mathscr{CL}$ into NL since the result of CLAN in case a conflict is detected is given as a (eventually huge) $\mathscr{CL}$ formula representing the counter-example.

The paper is organized as follows. In next section we recall the necessary technical background the rest of the paper is based on, including $\mathscr{CL}$ and GF. In section 3 we present our framework in general terms, and we provide some details on the implementation. Before concluding in the last section we present a case study in section 4.

2 Background

2.1 The Contract Language $\mathscr{CL}$

$\mathscr{CL}$ is a logic based on combination of deontic, dynamic and temporal logics, designed to specify and reason about legal and electronic (software) contracts [9, 10]. With the help of $\mathscr{CL}$ it is possible to represent the deontic notions of obligation, permission and prohibition, as well as the penalties applied in case of not respecting the obligations and prohibitions. In what follows we recall the syntax of $\mathscr{CL}$, and we give a brief intuitive explanation of its notations and terminology, following [10]. A contract in $\mathscr{CL}$ may be obtained by using the syntax shown in Fig. 1.

A contract clause in $\mathscr{CL}$ is usually defined by a formula C, 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, the obligation to perform an action α is written as $O_C(\alpha)$, illustrating the primary obligation to perform α, and if α is not performed then the reparation contract C is enacted. The above represents in fact a *CTD* (*Contrary-to-Duty*) as it specifies what is to be done if the primary obligation is not fulfilled. The prohibition to perform α is represented by the formula $F_C(\alpha)$, which not only specifies what is forbidden but also what is to be done in case the prohibition is violated (the contract C); this is called *CTP* (*Contrary-to-Prohibition*). Both CTDs and CTPs are then useful to represent normal (expected) behavior as well as the alternative (exceptional) behavior. $P(\alpha)$ represents the permission of performing a given action α. In the description of the

syntax, we have also represented what are the allowed actions (α and β in Fig. 1). It should be noticed that the usage of the Kleene star (the * operator) which is used to model repetition of actions, is not allowed inside the above described deontic modalities, though they can be used in dynamic logic style-conditions. Indeed, actions β may be used inside the dynamic logic modality (the bracket $[\cdot]$) representing a condition in which the contract C must be executed if action β is performed. The binary constructors &, ., and + represent concurrency, sequence and choice in basic actions (e.g. "buy", "sell") respectively. Compound actions are formed from basic actions by using the above operators. Conjunction of clauses can be expressed using the $\wedge$ operator; the exclusive choice operator ($\oplus$) can only be used in a restricted manner. $\top$ and $\bot$ are the trivially satisfied and violating contract respectively. 0 and 1 are two special actions that represent the impossible action and the skip action (matching any action) respectively.[2]

The following example is an excerpt from part of a contract between an 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", and "if the client does provide false information, the provider may suspend the service" [8]. If we consider the action *fi* (representing that "client supplies false information to Client Relations Department"), and *s* (representing that "provider suspends service"), in $\mathscr{CL}$ the above would be written as $F_{P(s)}(fi)$.

One of the usefulness of $\mathscr{CL}$ is that there are tools available to detect whether a given $\mathscr{CL}$ formula contains *conflicts*. There are four main kinds of conflicts in normative systems, and contracts in particular. The first one is when there is an obligation and the prohibition of performing the same action, in which case, independently of what the performed action is, will lead to a violation of the contract. The second conflict type happens when there is a permission and a prohibition to do the same action, which might lead to a contradicting situation. The other two cases are when there is an obligation to perform mutually exclusive actions, and the permission and the obligation to perform mutually exclusive actions. CLAN is a tool that automatically determines whether there are conflicts on $\mathscr{CL}$ formulae, giving a counter-example in case a conflict is detected [3].

2.2 Grammatical Framework

The Grammatical Framework (GF) is a framework to define and manipulate grammars [11]. Historically, GF was first implemented in the project "Multilingual Document Authoring" at Xerox Research Center Europe in Grenoble in 1998 [11]. As explained in [4] 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. The development of GF continued over time and evolved into a functional programming language, with multiple application domains [11]. GF has a central data structure called *abstract syntax*, based on type theory where it is possible to define special purpose grammars. It also has a *concrete syntax* part where it is possible to specify how the formulas defined in the abstract syntax can be translated into NL or a formal notation. One notable use of GF useful for our purposes, is the possibility to relate NL and formal languages in both directions: it is possible to go from a sentence written in NL to a term in the formal language, and vice-versa. GF has a module system consisting basically of two main modules, one to define the abstract syntax and the other the concrete syntax.

Abstract syntax. Abstract syntax is a type-theoretical part of GF where logical calculi as well as mathematical theories may be defined simply by using type signatures [4]. Let us consider the simple "Hello

[2] Some of the reasons behind the $\mathscr{CL}$ syntax have been motivated by a desire to avoid deontic paradoxes [6]. See [9] for a more detailed explanation of the design decisions behind $\mathscr{CL}$.

world" example. We start by defining the types of meaning (categories) `Greeting` and `Recipient`: `cat Greeting ; Recipient`.
We then define `Hello` as a function for building syntactic trees. The type of `Hello` has `Greeting` as its value type and `Recipient` as its argument type: `fun Hello: Recipient -> Greeting;`.
Concrete syntax. Once an abstract syntax is constructed, we can build a concrete syntax by using *linearization* rules, which basically allow us to generate the language. For instance, the English linearization rules for the function `Hello` is: `lin Hello recip = {s = ‘‘hello’’ ++ recip.s};`.
The above defines the linearization of `Hello` in terms of the linearization of its argument, represented by the variable `recip`. The terminal `‘‘hello’’` is concatenated with `recip.s` using the concatenation operator ++, which combines 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.

The example above shows that the `fun` rules should be defined in the abstract syntax modules and `lin` rules in concrete syntax modules. In the abstract syntax is where we need to define powerful type theory to express dependencies among parts of texts, and in the concrete syntax we need to define language-dependent parameter systems and the grammatical structures.

From the practical point of view, the concrete syntax description above could be saved on a file (let say `example.gf`), which could be uploaded by making `import example.gf`. By executing the command line `Parse "hello world"` we generate the abstract syntax `Hello World`. It should be noticed that `Hello` is the function defined in the abstract syntax above (prefixed with `fun`), and `Recipient` is the argument type of the function (i.e., `World` is of type `Recipient`). In the concrete syntax, `Recipient` is a record `recip` containing a string `s` which in this case will take the value `"world"`.

To summarize, the two main functionalities in GF useful for our purpose are *linearization* (translation from abstract to concrete syntax) and *parsing* (translation from concrete to abstract syntax).

3 Our framework

In this section we present our framework, AnaCon, in general terms, and we present a summary of the linearization and parsing process of $\mathscr{CL}$ into GF.

3.1 The framework in general terms

AnaCon takes as input a text file containing the description of a contract consisting of 3 parts: 1) A *Dictionary* listing the actions being used in the contract together with a textual description; 2) The contract itself written in restricted English; 3) A list of contradictory actions. (Fig. 2 shows the input file containing a simple contract.)

Our framework is summarized in Fig. 3 where arrows represent the flow of information highlighting how it works. Essentially, it consists of a parser, the Grammatical Framework, the conflict analysis tool CLAN, and some scripts used to connect these different parts. Overall, the typical workflow of AnaCon is as follows:

1. The user writes a contract (specification, set of requirements, etc) in NL (“plain” English), which is then manually translated into restricted English. This is a modeling task and it is done manually. It does not require any technical skill from the user, only to get access to the list of “allowed” English words to be used in the restricted version of the language. For instance, a sentence originally written as “The ground crew is obliged to open the check-in desk” would be translated into

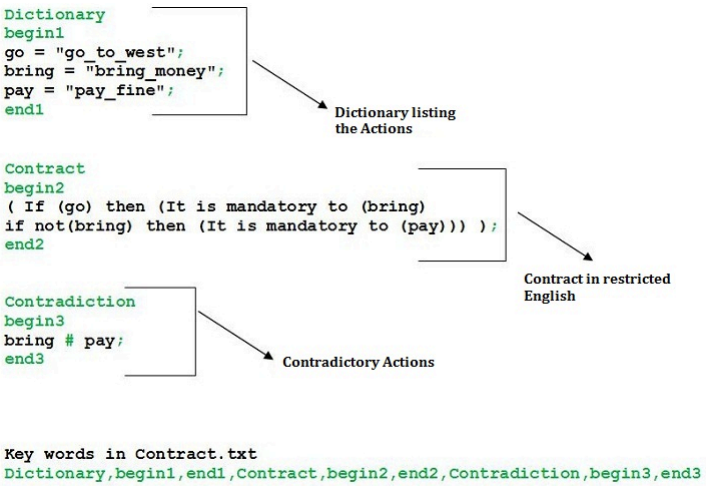

Figure 2: Sample Contract.txt

"It is mandatory to open the check-in desk", where "open the check-in desk" is an action name representing the real action.

2. The version of the contract written in restricted English is then passed to AnaCon script as an argument so the analysis starts (in what follows we call the input file containing the given contract, *Contract.txt*).
3. `Cont_ParserScriptGen` (a Java program) generates a script file based on the content of *Contract.txt*. The script file `Cont_Parser` then projects the content of the file to `testGrammarCl` parser.
4. At this stage `testGrammarCl` conducts syntax analysis based on the structure defined for the system. This parser is based on Labelled BNF grammar and generated from BNF Converter [5].
5. The Java program `Comparison` connects with `testGrammarCl` in order to obtain actions defined in the contract and then compare these actions against the ones defined in the *Dictionary* part of *Contract.txt*. Other analysis such as comparison between actions defined in *Contradiction* and the ones in *Dictionary*, duplication of actions in *Dictionary* and empty string assertion, are conducted at this level.
6. After successful parsing, the `Cont_GF_Cl` script file is generated with the contract and necessary information to start the translation process in GF from Restricted English to $\mathscr{CL}$.
7. The version of the contract written in restricted English is then represented in the abstract syntax part of GF.
8. The abstract syntax obtained above is translated into concrete syntax ($\mathscr{CL}$) which is then stored in the text file *Result_Cl.txt*.
9. The concrete syntax ($\mathscr{CL}$ formula) in textual form is transformed into XML by using `Cl2XML` (implemented but not integrated).
10. The XML version of the $\mathscr{CL}$ output of GF is fed into CLAN for analyzing whether the contract has a conflict.
11. If the output of CLAN is 'NO', then the answer is immediately given to the user. If the answer is 'YES' then the counter-example will be given by CLAN (a big $\mathscr{CL}$ formula containing the conflicting subclauses as well as the trace leading to such a state).
12. The formula obtained from CLAN is then linearized into a restricted English using GF. The `Cl_GF`

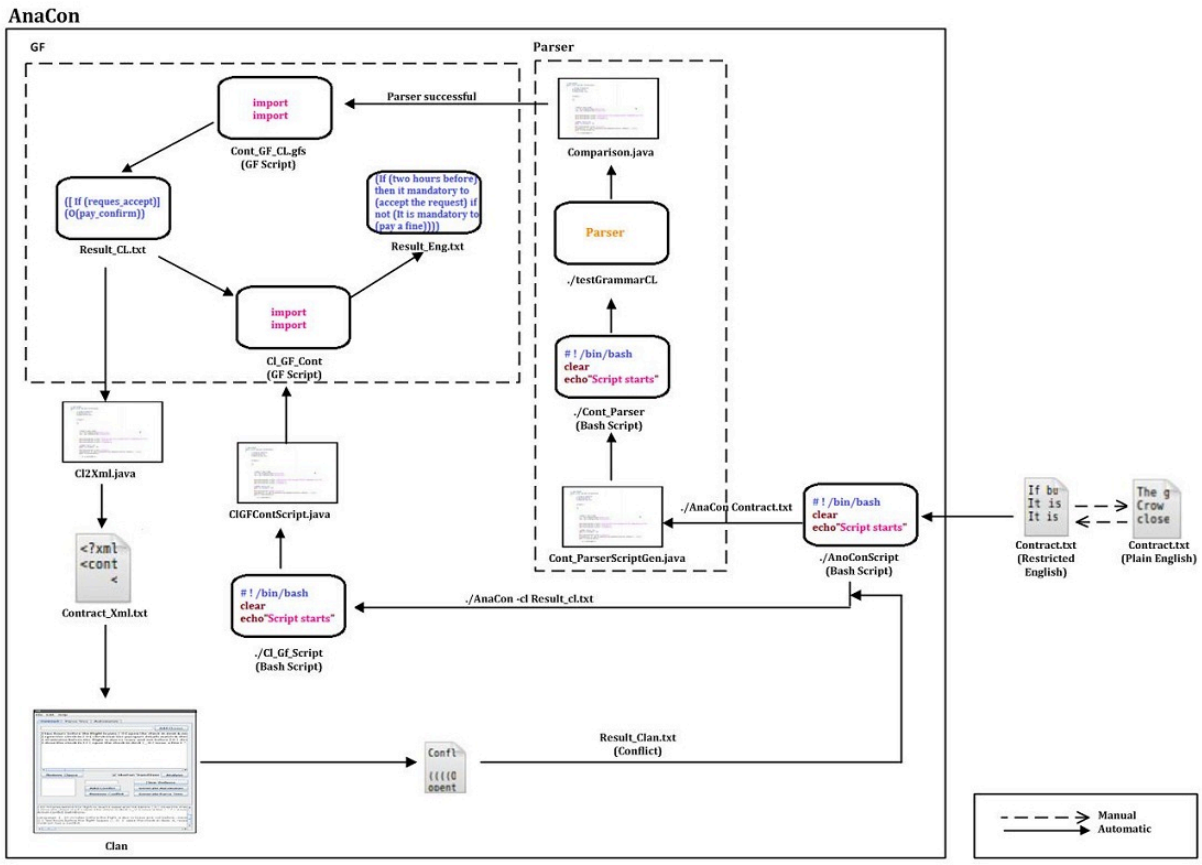

Figure 3: AnaCon Framework

script will take the content of *Result_Cl.txt* and pass it to `Cl_GF_ContScript` (a Java program), which generates the `Cl_GF_Cont` script to start the translation. The result in restricted English is given in a file named *Result_Eng.txt*.

13. The user must then find in the original contract where the counter-example arises. This last step is currently done manually, by simply searching in the text the keywords given in the counter-example in restricted English. (We discuss in the last section our future work on how to automate part of this process.)

Except for the steps above where we explicitly mention that it is manual, the rest of the process is completely automatic. So far, we have only fully implemented steps 2–8, and the translation back from $\mathscr{CL}$ to restricted English (as used in step 12).

Besides the above, we provide the possibility of generating a restricted English version of a $\mathscr{CL}$ formula, by executing AnaCon with a special flag (`AnaCon -cl <input_file.txt>`).

In what follows we will present some details of the implementation concerning the use of GF only.

3.2 Linearization and Parsing

In what follows we present the abstract and concrete syntax of $\mathscr{CL}$ and NL, in GF. At first, we present all the categories and functions for handling different $\mathscr{CL}$ clauses and actions in the abstract syntax part,

and then we concentrate on the concrete syntax part showing their representation in natural language. Due to lack of space we only show here some parts of the process.

The abstract syntax module as a central structure contains the basis for representation and formalization of $\mathscr{CL}$ syntax. The linearization of the different used functions and structures into $\mathscr{CL}$ symbolic syntax and natural language is done by using two concrete syntax modules. In the first one we write the exact $\mathscr{CL}$ syntax, and in the latter we express the corresponding restricted natural language.

We define the following categories, based on the BNF of $\mathscr{CL}$.

```
 --  Abstract module(Cl.gf module)
cat    Act ;KleeneStarAct;KleeneCompAct;ClSO;ClSF;ClSP;Clause;Clauses;ClauseP;
ClauseO; ClauseF;And;Or;Dot;Cross;CompAct;Star;Not;[Clause]{2};[CompAct]{2};
[ClauseO]{2}; [ClauseP]{2};
```

We define similar categories in the concrete modules `ClEng.gf` and `ClSym.gf`, using `lincat` instead of `cat` prefixing them. The main difference is that in the linearization type definitions we need to state that for instance `Clause` and `Act` are records containing a field of type string `s`.

Obligations, Permissions and Prohibitions. Obligations, permissions and prohibitions have the same structure in the `Cl.gf` module. In the rest of this section we will only show the abstract and concrete syntax for obligations, those for permissions and prohibitions being similar.

```
 --  Abstract module(Cl.gf module)
fun Obl : ClSO -> CompAct -> Clauses;
    OClause : ClauseO -> Clause;;
    Clo : ClSO -> CompAct -> ClauseO;
```

There are some differences among these clauses (or rather group of clauses, as there are similar ones for permissions and prohibitions) which is worth mentioning. As it is clearly shown `CompAct` is the argument type used among the four groups which represent both basic and compound actions respectively. This will provide the possibility to be able to express `Obligation` over both basic and compound actions (actions will be explained later on in this section). `ClauseO` follows the BNF of the syntactic definition of $\mathscr{CL}$ obligations and allows to express the obligation that together with the action (verb) form the actual clause. The other difference is that generally a `Clause` itself can consist of different structures and thus it can be either constructed from `ClauseO`, `ClauseP` and `ClauseF` from basic or compound actions. They are defined in this way to avoid mixing certain operators only permitted for some of the deontic notions. It facilitates to make the conjunction and choice of certain kind of clauses, but not the direct linearization and parsing of each structure. For that we specify `Clauses` as a start category for parsing and linearization so that each structure can be linearized and parsed directly. The linearization of the above in concrete syntax is as follows:

```
 --  Concrete module(ClEng.gf module)
lin  Clo clo compact  = {s = "(" ++ clo.s ++ "(" ++ compact.s ++ ")" ++ ")"};
     OClause clo = {s = clo.s};
     Obl clo compact  = {s = "(" ++ clo.s ++ "(" ++ compact.s ++ ")" ++ ")"};
```

Basically, `Clo` and `Obl` clauses are expressed in NL using restricted words as “It is mandatory to” (`clo.s`) as terminals (quoted words in GF are *terminals*) so that together with actions they formulate the clauses in NL.

As explained before, we provide another concrete syntax module called `ClSym.gf` to provide the user with the possibility of writing with specific $\mathscr{CL}$ syntax, such as operators, parentheses, brackets,

etc. The converse is also true, when writing any specific clause in NL, the framework (with the help of this module) would be able to provide $\mathscr{CL}$ formulas in the intended format.

```
 --  Concrete module(ClSym.gf module)
lin  Clo clo compact  = {s = "(" ++ clo.s ++ "(" ++ compact.s ++ ")" ++ ")"};
     OClause clo = {s = clo.s};
     Obl clo compact  = {s = "(" ++ clo.s ++ "(" ++ compact.s ++ ")" ++ ")"};
```

The structure of the above module is very similar to the `ClEng.gf` module, the only difference being that `clo.s` represent specific characters such as `"O"` instead of words or sentences.

```
 --  Concrete module(ClSym.gf module)
lin O = {s = "O"};
```

Contrary-to-Duties (CTDs) and Contrary-to-Prohibitions (CTPs). CTDs and CTPs are both related to obligation and prohibition clauses respectively, and are expressed as the following functions:

```
 --  Abstract module(Cl.gf module)
fun  CTDc :  CompAct -> Clause -> Clauses;
     CTDcc: CompAct -> Clause -> Clause;
```

CTD and CTP clauses are functions taking an action (which is to be obliged, or prohibited) and a clause representing what is to be done in case the obligation or the prohibition is not fulfilled. We specify again these operators over simple and compound actions. We only present now the concrete modules for CTDs, the ones for CTPs being similar.

```
 --  Concrete module(ClEng.gf module)
lin CTDc compact clause = {s = "(" ++ "It is mandatory to" ++ "(" ++ compact.s ++ ")"
     ++"if not" ++ "(" ++ compact.s ++ ")"++ "then" ++ clause.s ++ ")"};

    CTDcc compact clause = {s = "(" ++ "It is mandatory to" ++ "(" ++ compact.s ++ ")"
      ++ "if not" ++ "(" ++ compact.s ++ ")" ++ "then" ++ clause.s ++ ")"};
```

Now, we show how to express the logical syntax in the other concrete module:

```
 --  Concrete module(ClSym.gf module)
lin CTDc compact clause = {s = "(" ++ "O" ++ "(" ++ compact.s ++ ")" ++ "_"
     ++ clause.s ++ ")"};
    CTDcc compact clause = {s = "(" ++ "O" ++ "(" ++ compact.s ++ ")" ++ "_"
    ++ clause.s ++ ")"};
```

The only important thing to notice here is the ‘‘_’’ character 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 primary obligation.

Conjunction of clauses ($C \wedge C$) Other operators, like conjunction and exclusive or, need also to be represented in the abstract and concrete syntax. We only show here how to represent the conjunction.

```
 --  Abstract module(Cl.gf module)
fun Conj_np : [Clause] -> Clauses;
    Conj_np2 : [Clause] -> Clause;
```

As it is clearly defined in the above representation the structure used for conjunction of clauses consists of list of clauses which may be any kind of clause such as obligation, prohibition, etc. In this it is possible to define conjunction of many clauses. The two concrete modules are as follows:

```
 --  Concrete module(ClEng.gf module)
lin Conj_np xs = {s = "(" ++ xs.s ! Conjunction_np ++ ")"};
    Conj_np2 xs = {s = "(" ++ xs.s ! Conjunction_np ++ ")"};

 --  Concrete module(ClSym.gf module)
lin Conj_np xs = {s = "(" ++ xs.s ! Conjunction_np ++ ")"};
    Conj_np2 xs = {s = "(" ++ xs.s ! Conjunction_np ++ ")"};
```

The structure used in above to show iteration conjunction of clauses, corresponds to the way it was defined in $\mathscr{CL}$.

Test Operator. The test operator where in $\mathscr{CL}$ is used to express conditional obligations, permissions and prohibitions. The test operator may be applied to simple or compound actions including the Kleene star. We should thus add a function to each different application of the test operator; we will only present the abstract and concrete syntax of the application of the Kleene star to simple and compound actions.

```
 --  abstract module(Cl.gf module)
fun TestOpc,TestOpcStar : KleeneCompAct -> Clause -> Clauses;
    TestOpcc,TestOpccStar : KleeneCompAct -> Clause -> Clause;
```

The concrete modules are as follows:

```
 --  concrete module(ClEng.gf module)
  fun TestOpc kleenecompact clause = {s = "(" ++ "If" ++ "(" ++ kleenecompact.s ++")"
    ++ "then"++ clause.s ++ ")"} ;

  TestOpcStar kleenecompact clause = {s ="("++"((" ++ "Always" ++ ")"|"(" ++ "After"
    ++ ")"|"(" ++ "When" ++ ")"|"(" ++ "Before" ++ ")") ++ "(" ++ "If" ++ "(" ++
    kleenecompact.s ++ ")" ++ "then"++ clause.s ++ ")" ++ ")"} ;

  TestOpcc kleenecompact clause = {s = "(" ++ "If" ++ "(" ++ kleenecompact.s ++ ")"
    ++ "then"++ clause.s ++ ")"};

   TestOpccStar kleenecompact clause = {s = "(" ++ ( "(" ++ "Always" ++ ")"|"(" ++
    "After" ++ ")"|"(" ++ "When" ++ ")"| "(" ++ "Before" ++ ")") ++ "(" ++ "If" ++
    "(" ++ kleenecompact.s ++ ")" ++  "then"++ clause.s ++ ")" ++ ")"};

 --  concrete module(ClSym.gf module)
   fun TestOpc kleenecompact clause = {s = "(" ++ "[" ++ "(" ++ kleenecompact.s ++")"
     ++ "]" ++ clause.s ++ ")"} ;

   TestOpcStar kleenecompact clause = {s = "(" ++ "[" ++ "1" ++ "*" ++ "]" ++
    "(" ++ "[" ++ "(" ++ kleenecompact.s ++ ")" ++ "]" ++ clause.s ++ ")" ++ ")"} ;

   TestOpcc kleenecompact clause = {s = "(" ++ "[" ++ "(" ++ kleenecompact.s
   ++ ")" ++ "]" ++ clause.s ++ ")"};

   TestOpccStar kleenecompact clause = {s = "(" ++ "[" ++ "1" ++ "*" ++ "]"
    ++ "(" ++ "[" ++ "(" ++ kleenecompact.s ++ ")" ++ "]" ++ clause.s ++ ")"++ ")"};
```

Actions. Defining basic, compound and Kleene star actions in GF is not difficult, 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), it is more efficient to use module extension [11]. This in effect separates the grammar part (`Cl.gf` module) from a more specific vocabulary part (`Action` module). In other words, the developer will be provided with a modular system giving more flexibility to modify the modules, and thus increasing maintainability. The `Action` module extends the `Cl` module. In such a module we will define all the involved simple actions (e.g., "Pay), other actions only affected by the Kleene star (e.g., "CloseCheckIn"), as well as those operators over actions. In what follows we only show the conjunction (sequence of actions, choice, etc are defined similarly).

```
 --  Action module(abstract syntax)
fun Pay,Buy : Act;

 --  Cl module (abstract syntax)
fun CompActSI : Act -> CompAct;
    CompActa : CompAct -> And -> CompAct -> CompAct;
```

The linearization of the functions specified in the `Action` module above described is easy to specify:

```
-- ActionEng module (concrete syntax)
lin Pay = {s = "pay a fine"};
    Buy = {s = "buy a car"};
    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 on in the translation of compound actions into NL is to know how each operator should be interpreted. 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 CompActSI : Act -> CompAct;
    CompActa : CompAct -> And -> CompAct -> CompAct;

-- ClEng module (concrete syntax)
lin CompActSI acti  = {s = acti.s};
    CompActa compact and compact1 = {s = compact.s ++ and.s ++ compact1.s};
```

User defined operations such as the above fall under specific logical symbols which are defined below:

```
-- ClSym module (concrete syntax)
lin  CompActa CompAct and CompAct1 = {s = CompAct.s ++ and.s ++ CompAct1.s};
     CompActSI acti  = {s = acti.s};
```

In this manner, the representation of `and.s` (similarly for `or.s` , `dot.s`, and `not.s`) are reduced to `"&"`, `"+"`, `"."` and `"!"` which are logical operators as used in CLAN to manipulate $\mathscr{CL}$ formulae.

The Kleene star is actually a compound action as shown below with the difference that it can only be used between test operator:

```
-- Cl module (abstract syntax)
fun  KleeneActSI : Act -> KleeneCompAct;
     KleeneActa : KleeneCompAct -> And -> KleeneCompAct -> KleeneCompAct;
```

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.*

Figure 4: Case study

In what follows we show the corresponding concrete syntax enabling the translation to natural and symbolic languages:

```
-- ClEng module (concrete syntax)
lin  KleeneActSI acti = {s = acti.s};
     KleeneActa kleenecompact and kleenecompact1  = {s = kleenecompact.s ++
and.s ++ kleenecompact1.s};

-- ClSym module (concrete syntax)
lin  KleeneActSI acti = {s = acti.s};
     KleeneActa kleenecompact and kleenecompact1  = {s = kleenecompact.s
++ and.s ++ kleenecompact1.s};
```

4 Case Study

In this section we apply our framework to a case study taken from [2] of a contract concerning the check-in process of an airline company. The full description is given in Fig. 4. We provide the detailed translation into restricted English, and the corresponding $\mathscr{CL}$ formula for all the clauses. Note that clause 10 is "distributed" among the others as it represent a penalty in case the other clauses are not satisfied.

1. The ground crew is obliged to open the check-in desk and request the passenger manifest two hours before the flight leaves (Fig. 4 first clause).

[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 (pay_a_fine))))

[Program output]: ([(two_hours_before_the_flight_leaves)] (O (open_the_check_in_desk & request_the _ passenger_manifest) _ (O (pay_a_fine))))

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

   [Restricted English]: ((When) (If (opening_the_desk_with_the_passenger_manifest) then (It is mandat ory to (reply_to_the_passenger_manifest_request) if not (reply_to_the_passenger_manifest_request) then (It is mandatory to (pay_a_fine)))))

   [Program output]: ([1 *] ([(opening_the_desk_with_the_passenger_manifest)] (O (reply_to_the_ pass enger_manifest_request) _ (O (pay_a_fine)))))

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 (Fig. 4 third clause).

   [Restricted English]: (((After) (If (open_the_check_in_desk) then (It is mandatory to (check_that_the _ passport_details_match_ what_is_written_on_the_ticket_and_check_ the_luggage_is_within_the_ weight_limits) if not (check_that_the_passport_details_match_what_is_written_on_the_ticket_and_ check_the_luggage_is_within_the _weight_limits) then (It is mandatory to (pay))))) and (If (check_ that_the_passport_details_match_what_is_written_on_the_ticket_and_check_the_luggage_is_within _the_weight_limits) then (It is mandatory to (issue_the_boarding_pass) if not (issue_the_boarding_ pass) then (It is mandatory to (pay_a_fine)))))

   [Program output]: (([1 *] ([(open_the_check_in_desk)] (O (check_that_the_passport_details_ match_what_is_written_on_the_ticket & check_the_luggage_is_within_the_weight_limits) _ (O (pay_ a_fine))))) ∧ ([(check_that_the_ passport_details_match_what_is_written_on_the_ticket & check_ the_luggage_is_within_the_weight_limits)] (O (issue_the_boarding_pass) _ (O (pay_a_fine)))))

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 (Fig. 4 seventh clause).

   [Restricted English]: (If (the_luggage_weighs_more_than_the_limit) then (It is mandatory to (colle ct_ payment_for_the_extra_weight_and_issue_the_boarding_pass) if not (collect_payment_for_the_ extra_weight_ and_issue_the_boarding_pass) then (It is mandatory to (pay_a_fine))))

   [Program output]:([(the_luggage_weighs_more_than_the_limit)] (O (collect_payment_for_the_ extra_weight & issue_the_boarding_pass) _ (O (pay_a_fine))))

5. The ground crew is prohibited from issuing any boarding cards without inspecting that the details are correct beforehand. (Fig. 4 ninth clause).

   [Restricted English]: (It is mandatory to (inspect_that_the_details_are_correct_beforehand) if not (inspect_that_the_details_are_correct_beforehand) then (It is prohibited to (issue_any_boarding_cards) if (issue_any_boarding_cards) then (It is mandatory to (pay_a_fine))))

   [Program output]:(O (inspect_that_the_details_are_correct_beforehand) _ (F (issue_the_boarding_pass) _ (O (pay_a_fine))))

6. The ground crew is prohibited from issuing any boarding cards before opening the check-in desk (Fig. 4 tenth clause).

   [Restricted English]: ((Before) (If (open_the_check_in_desk) then (It is prohibited to (issue_any_boarding_cards) if (issue_any_boarding_cards) then (It is mandatory to (pay_a_fine)))))

   [Program output]: ([1 *] ([(open_the_check_in_desk)] (F (issue_the_boarding_pass) _ (O (pay_a_fine)))))

7. 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]:((Before) (If (20_minutes_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 (pay_ a_fine)))))

   [Program output]:([1 *] ([(20_minutes_the_flight_is_due_to_leave_and_not_before)] (O (close_the_check_in desk) _ (O (pay_a_fine)))))

8. After closing check-in, the crew must send the luggage information to the airline

   [Restricted English]:((After) (If (close_the_check_in_desk) then (It is mandatory to (send_the_luggage_information_to_airline) if not (send_the_luggage_information_to_airline) then (It is mandatory to (pay_a_fine)))))

   [Program output]:([1 *] ([(close_the_check_in_desk)] (O (send_the_luggage_information_to_airline) _ (O (pay_a_fine)))))

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

   [Restricted English]:((Always) (If (close_the_check_in_desk) then (It is prohibited to (issue_any_boarding_pass_or_open_the_check_in_desk) if (issue_any_boarding_pass_or_open_the_check_in_desk) then (It is mandatory to (pay_a_fine)))))

[Program output]:([1 *] ([(close_the_check_in_desk)] (F (issue_the_boarding_pass + open_the_check_in_desk) _ (O (pay_a_fine)))))

The case above has already been analyzed using CLAN before and a conflict has been detected as reported in [3]. So, in this sense we do not report any new result here. We have used the same example as our intention is to validate our approach on a familiar case study when all the steps in our framework be implemented (we are currently working on a full implementation of the framework).

5 Related Work

GF has been used on a variety of application domains. We will only focus here on the one reported in [4] since it is closely related to our research. In such paper Hähnle et al. describes how to get a NL version of a specifications written in OCL (Object Constraint Language). The paper focused on helping to solve problems related to authoring well-formed formal specifications, maintaining them, mapping different levels of formality and synchronizing them. The solution outlined in the paper illustrates the feasibility of connecting specification languages at different levels, in particular OCL and NL. The authors have implemented different concepts of OCL such as classes, objects, attributes, operations and queries in GF. Our work is similar to [4] with the difference that $\mathscr{CL}$ is a more abstract and general logic allowing to specify contracts in a general sense (as mentioned in the introduction $\mathscr{CL}$ may formalize legal contracts, software specifications, contracts in SOA, or even be used to represent requirements). Besides we are not interested only on "language translation" but rather in the use of the formal language to further perform verification (in our case conflict analysis) which is then integrated within our framework by connecting GF's output into CLAN. In what concerns the technical difficulties related to the implementation in GF we do not have enough knowledge of the work done in [4] as to make a more careful comparison.

From the perspective of relating a contract language and natural language, it is worth mentioning the work by Pace and Rosner [7], where it is presented an end-user system which is specifically designed to process the domain of computer oriented contracts. The translation is not based on GF but on a completely different technology. They use controlled natural language (CNL) to specify contracts, and define a similar logic to $\mathscr{CL}$, which is embedded into Haskel in order to manipulate the contracts. So, the comparison is not straightforward as the aim of their work and ours diverges.

6 Conclusions

We have presented in this paper an encoding of the contract language $\mathscr{CL}$ into GF, and back. We have integrated the above into a framework that allows to analyze $\mathscr{CL}$ formulae for conflicts, and eventually give a counter-example in restricted English helping the user to find it in the original specification in natural language. As a proof-of-concept we have applied it to a case study which has already been used for conflict detection. The framework as presented here does not automate the whole process, though we are working on those parts as described below.

We would like to emphasize what was said in the introduction concerning the scope of our approach. $\mathscr{CL}$ is a formal language to specify contracts in a broad sense, and as such one should not think that our work limits to the analysis of contracts in that language. As an abstract logic, $\mathscr{CL}$ can be used to describe and prescribe "contracts" (including specifications) in SOA, component-based development systems, e-business, requirement engineering, etc. We believe the approach is useful in practice, the

only potential bottleneck being CLAN since the current version is not optimized as to obtain small non-redundant automata (the tool is very much a specialized explicit model checker, where high number of transitions are generated due to the occurrence of concurrent actions).

One practical way to reduce the size of the automaton created by CLAN is to try to define as many mutually exclusive actions as possible. Note that some of the actions in our contract example are obviously mutually exclusive (e.g., 'open the check in desk' and 'close the check in desk'), while others are mutually exclusive in the "formal" sense, that is we know that they cannot occur at the same time (for instance, 'issue a fine' and 'issue the boarding pass'). We are currently working on more fundamental ways to improve the performance of CLAN by reducing the size of the automaton while building it.

A challenging future work concerns the use of Passage Retrieval tools (as for instance the one presented in [1]) to help to find the counter-example in the original English contract by using the information in restricted English (obtained from CLAN and translated into English by our framework). This will avoid to manually go through big part of the English text, increasing efficiency and precision. Another interesting line of research is to study how to combine our approach to the one presented in [7]. We believe we could then improve our analysis capability, by using specialized tools for some purposes (as we have done here, using CLAN), and the embedded language technology for others.

References

[1] D. Buscaldi, P. Rosso, J.M. Gómez-Soriano & E. Sanchis (2009): *Answering Questions with an n-gram based Passage Retrieval Engine*. Journal of Intelligent Information Systems 34(2), pp. 113–134. doi:10.1007/s10844-009-0082-y.

[2] S. Fenech, G.J. Pace & G. Schneider (2009): *Automatic Conflict Detection on Contracts*. In: *ICTAC'09*, *LNCS* 5684, Springer, pp. 200–214. doi:10.1007/978-3-642-03466-4_13.

[3] S.Fenech, G.J. Pace & G. Schneider (2009): *CLAN: A Tool for Contract Analysis and Conflict Discovery*. In: *ATVA'09*, *LNCS* 5799, Springer, pp. 90–96. doi:10.1007/978-3-642-04761-9_8.

[4] R. Hähnle, K. Johannisson & A. Ranta (2002): *An Authoring Tool for Informal and Formal Requirements Specifications*. In: *FASE*, *LNCS* 2306, Springer, pp. 233–248. Available at `http://link.springer.de/link/service/series/0558/bibs/2306/23060233.htm`. doi:10.1007/3-540-45923-5_16.

[5] M. Forsberg A. Ranta. M. Pellauer (2004-09): *BNF Converter: Multilingual Front-End Generation from Labelled BNF Grammars*. Technical Report, Technical Report 348, Computing Science at Chalmers University of Technology and Gothenburg University.

[6] P. McNamara (2006): *Deontic Logic*. In: *Gabbay, D.M., Woods, J., eds.: Handbook of the History of Logic*, 7, North-Holland Publishing, pp. 197–289.

[7] G.J. Pace & M. Rosner (2010): *A Controlled Language for the Specification of Contracts* 5972, pp. 226–245. doi:10.1007/978-3-642-14418-9_14.

[8] G. Pace, C. Prisacariu & G. Schneider (2007): *Model Checking Contracts –A Case Study*. In: *ATVA'07*, *LNCS* 4762, Springer-Verlag, pp. 82–97. doi:10.1007/978-3-540-75596-8_8.

[9] C. Prisacariu & G. Schneider (2007): *A Formal Language for Electronic Contracts*. In: *FMOODS*, *LNCS* 4468, Springer, pp. 174–189. doi:10.1007/978-3-540-72952-5_11.

[10] C. Prisacariu & G. Schneider (2009): *CL: An Action-based Logic for Reasoning about Contracts*. In: *WOLLIC'09*, *LNCS* 5514, Springer, pp. 335–349. doi:10.1007/978-3-642-02261-6_27.

[11] A. Ranta (2009): *Grammatical Framework: A Programming Language for Multilingual Grammars and Their Applications*. `http://www.cs.um.edu.mt/svrg/Tools/CLTool`.