Formal Verification of Multilingual Instructions

Kristofer Johannisson and Aarne Ranta
Department of Computing Science
Chalmers University of Technology and Göteborg University

Abstract

This paper proposes the idea of applying the methods of program verification to texts written in natural language. The idea is illustrated by a small real-world example, the instructions for using an alarm system. We will formalize the fragment of English used in the instructions, define the semantic notion of correctness, and formally prove that an actually used set of instructions is correct with respect to the semantics. We will also construct a machine-translation system for this type of instructions for four languages.

The instruction grammars are defined in the grammar formalism GF. The proof is carried out in the proof editor Alfa, into which GF can be plugged in, by virtue of a common underlying type theory.

1 Introduction

1.1 The correct functioning of a human-machine system

Consider a complex system based on the interaction of human beings and machines, such as a nuclear powerplant. There is a physical process controlled by human employees using machines—mechanical and electronic. The reliability of the process depends partly on the functioning of the machines, partly on the actions performed by the humans. The machines get instructions from computer programs, as well as directly from humans. The humans get instructions from technical manuals, as well as from machines (light signals, monitors).

It is vital for the functioning and security of the nuclear powerplant both that the instructions are properly followed and that the instructions are correct. In the situation that we just sketched, there are four relevant points:

1) **Function correctness**: the functions defined by controlling programs must be properly executed by the machinery.

2) **Program correctness**: the programs must define correct and safe functions.

3) **Action correctness**: the humans must understand the instructions they read and act in accordance with them.

4) **Instruction correctness**: the technical manuals read by the humans must give correct and safe instructions.

Notice that this is an interdependent complex. Much of human intervention is necessitated precisely by the fragility of point (1): a bolt may get loose, preventing the machinery from functioning in the proper way. This demands an action by a technician, an action specified in his technical manual.

But it is possible to consider the four points in separation. Taking (1) and (2) for granted, let us consider first (3) and then (4). Let us assume that the staff must always follow the instructions literally—at least, they should never be sanctioned for doing so. Let us also assume that the staff need not take any initiatives, but that their actions are exhaustively described in their manuals. So the problem of action correctness reduces to the question whether the instructions are understood.

1.2 Controlled language

Controlled language is one solution to the intelligibility problem. The manuals are written in a certain way, purified from ambiguities and vaguenesses. It is only a small fragment of real language. For instance, in controlled languaged it may be forbidden to write

remove the battery with a screwdriver and replace it by a new one

You may be obliged to say

remove the battery with a screwdriver and replace the battery by a new battery

to prevent the interpretation as

remove the battery with a screwdriver and replace the screwdriver by a new screwdriver

The simple use of *it*, although obvious to any "normal" employee, could, for one just blindly following the formal rules of the language, produce embarrassment or even misunderstanding.

Controlled languages have a firm tradition of being used in aeronautic industry (cf. Lux [5]) but one can well see how the idea applies to any situation where precise instructions are given. The sense in which a language is controlled varies a lot. In one end—which is also the historical origin—we have **manuals for technical writing**, which give instructions on terminology, style, and perhaps syntax. More control is added to this by **language checkers**, which are computer programs that check whether given texts conform to given sets of rules. The definition of such a checker already presupposes a precise definition of the rules of the language, and in the other end of the scale we have a **completely specified language**, a language defined by a formal syntax. Nothing is an expression of such a language unless generated by the formal syntax.

Controlled languages currently used are mostly just partially defined, rather heterogeneous principles. The principles are typically negative in form: e.g. "do not use pronouns", "do not use the infinitival phrases to replace sentences". But let us concentrate on the distilled notion of completely specified language, defined in purely positive terms. In linguistics, such a language is recognized as one that results from a **generative grammar**: an inductive definition that generates all expressions of the language (in linguistics, this notion dates back to Chomsky [2]). Although the expressions of such a language are indistinguishable from expressions of natural language, they in fact belong to a certain artificial language, just as, for instance, the expressions of a programming language. The situation can be illustrated by a familiar example,

$$2 + 3 = 5.$$

This is, at the first sight, an expression of ordinary mathematical language, which we learn at school or even before, and which is not governed by explicitly stated syntactic rules. At the same time, it is a well-formed expression of many programming languages, but not of all of them. A part of the effort put in the design of programming languages has been to make such familiar mathematical notations available, but this has usually been only achieved in part.

To sum up, the ultimate conception of a controlled language makes it into something similar to programming languages. It is, by design, free from unclarities. The task of making it intelligible to human beings is analogous to the task of writing a compiler. This is sometimes guaranteed by the ordinary linguistic competence of the persons, but special training may well be needed in addition, to "translate" the specialized instructions of the controlled language into the more elementary instructions that the trainee can already be assumed to master.

1.3 Program and instruction correctness

Given a sound system of hardware, an adequate programming language, and a sound compiler, the correct functioning of a computer system depends entirely on the correctness of software. As the software is written in a programming language that is defined by formal rules, it is, in principle, possible to perform formal checks on it and verify that it really does certain things and does not do certain other things. The discipline concerning this is **program verification**, and there are several techniques of program verification developed under decades. Ideally, the correctness of a program can be established by a mathematical proof. This is the line we will follow in this paper, using constructive type theory as described in Nordström & al. [6].

Program verification is currently seldom applied to the whole of a large-scale industrial program, because it is expensive. But there are more and more programs whose proper functioning is so vital that the cost of verification is affordable. At the same time, the methods have improved. There is more and more demand for program verification in industry, and more and more work to satisfy the demand.

Given a complex system, such as a nuclear powerplant, whose functioning depends on machines and humans together, program verification is not enough. One should also eliminate "human error". This is, of course, not possible in full, but one can consider eliminating errors resulting from badly understood or unsound instructions. Controlled language is a solution to the understanding problem. The soundness of instructions, if written in a formally specified language, is a question perfectly analogous to program verification.

In the rest of this report, we will explain a method of building controlled languages in a way that permits the application of program verification techniques to instructions. We will consider both computational and linguistic aspects of this method: these aspects are related, since a completely formalized language is what makes it possible both to verify texts and translate them automatically into different languages.

2 The example

Rather than explaining the underlying theory in detail, we will present and discuss a complete real-world application in English, French, German, and Swedish. Although the application is real-world, it is so small that it would perhaps not be strictly necessary to use formal methods to convince oneself of the functioning of the system. But many of those who have learnt these instructions must have mentally performed a reasoning that shows that the system works.

The example is the following: the industrial research centre where one of the authors was once employed has an alarm system switched on during nights and weekends. It reacts to the presence of any moving body in the premises, and calls the security staff. Employees working in the research centre have the right to enter the building even in the alarmed hours by using their magnetic cards. At the entrance, however, they must first switch off the alarm, if it is on. When leaving the building, they must switch the alarm back on, if there is no-one left in the building. The employees are informed on the state of the alarm and the occupation of the building by two-coloured lights and by a whiteboard. A red light tells that the alarm is on, and a green light that it is off. The employees are in the possession of a secret code that they enter to switch the state of the alarm. On the whiteboard, there is a list of the names of those who are in. The employees are given the following instructions:

> *When entering the building: if the light is green, write your name on the board and then enter the premises. If the light is red, enter the code, then write your name on the board and then enter the premises.*
>
> *When leaving the building: erase your name from the board. If the board is empty, enter the code and then leave the premises. If the board is not empty, just leave the premises.*

We will perform two tasks on this text:

> Translate the text into three other languages, so that employees of different nationalities can read it in their own languages.
>
> Verify the soundness of the text: if everyone follows the instructions, the building cannot be empty while the alarm is off, neither can the alarm be on when there is someone in the building.

3 Translations

The English instruction text above was produced by automatic generation from a type-theoretical object, by using GF (Grammatical Framework, Ranta [7]). A GF "program", a grammar, consists of two parts: an **abstract syntax**, which is similar to a theory in logical frameworks, and a **concrete syntax**, which maps objects in the abstract syntax into an informal language. As regards the abstract syntax, GF has a common subset with the logical framework Agda/Alfa (Coquand [3], Hallgren [4]) which makes it possible to work on GF grammars in Alfa: the verification part of our present task has been performed in Alfa.

The following piece of GF code (translated into LaTeX by a GF utility) belongs to the abstract syntax of the instruction grammar:

> **cat** Text ; Instr ; Act ; Acts ; Cond ; Sent
>
> **fun** cond : (C : Cond) → (A : Acts) → Instr
>
> **fun** GoOut : Act
>
> **fun** AJust : (A : Act) → Acts
>
> **fun** CAtom : (S : Sent) → Cond
>
> **fun** NonEmp : Sent

The concrete syntax for English consists of **linearization rules** for each function constant of the abstract syntax. These rules produce records with a string and generally (although not in this case) some other linguistic information.

$$\begin{aligned}
&\textbf{lin}\ \text{cond C A} &&= \{\text{s} = \textit{if}\ {+}{+}\ \text{C.s}\ {+}{+}\ ,\ {+}{+}\ \text{A.s}\}\\
&\textbf{lin}\ \text{GoOut} &&= \{\text{s} = \textit{leave}\ {+}{+}\ \textit{the}\ {+}{+}\ \textit{premises}\}\\
&\textbf{lin}\ \text{AJust A} &&= \{\text{s} = \textit{just}\ {+}{+}\ \text{A.s}\ \}\\
&\textbf{lin}\ \text{CAtom S} &&= \{\text{s} = \text{S.s}\}\\
&\textbf{lin}\ \text{NonEmp} &&= \{\text{s} = \textit{the}\ {+}{+}\ \textit{board}\ {+}{+}\ \textit{is}\ {+}{+}\ \textit{not}\ {+}{+}\ \textit{empty}\}
\end{aligned}$$

For instance, the type-theoretical object of type **Instr**

cond (CAtom NonEmp) (AJust GoOut)

corresponds to the English string

if the board is not empty, just leave the premises.

In the Swedish rules, there is a subtle stylistic difference from English: the word *just* in front of an **Act** expression is translated by the phrase *helt enkelt*, which appears in the middle of the **Act** expression. (German, as shown below, is similar to Swedish). The GF grammar accomplishes this by decomposing the linearization of an **Act** expression into two discontinuous parts, s_1 and s_2.

$$\begin{aligned}
&\textbf{lincat}\ \text{Act} &&= \left\{\begin{array}{lcl}\text{s}_1 & : & \text{Str}\\ \text{s}_2 & : & \text{Str}\end{array}\right\}\\
&\textbf{lin}\ \text{cond C A}\ ; &&= \{\text{s} = \textit{om}\ {+}{+}\ \text{C.s}\ {+}{+}\ ,\ {+}{+}\ \text{A.s}\}\\
&\textbf{lin}\ \text{GoOut} &&= \left\{\begin{array}{lcl}\text{s}_1 & = & \textit{lämna}\\ \text{s}_2 & = & \textit{lokalerna}\end{array}\right\}\\
&\textbf{lin}\ \text{AJust A} &&= \{\text{s} = \text{A.s}_1\ {+}{+}\ \textit{helt}\ {+}{+}\ \textit{enkelt}\ {+}{+}\ \text{A.s}_2\}\\
&\textbf{lin}\ \text{CAtom S} &&= \{\text{s} = \text{S.s}\}\\
&\textbf{lin}\ \text{NonEmp} &&= \{\text{s} = \textit{tavlan}\ {+}{+}\ \textit{inte}\ {+}{+}\ \textit{är}\ {+}{+}\ \textit{tom}\ \}
\end{aligned}$$

The Swedish version of the instructions, generated by automatic translation from the English text in GF, is

> *Då du går in i huset: Om lampan är grön, skriv ditt namn på tavlan och gå därefter in i lokalerna. Om lampan är röd, tryck in koden, skriv därefter ditt namn på tavlan och gå därefter in i lokalerna.*
>
> *Då du lämnar huset: tag bort ditt namn från tavlan. Om tavlan är tom, tryck in koden och lämna därefter lokalerna. Om tavlan inte är tom, lämna helt enkelt lokalerna.*

The French and German version, also translated from English, are

> *En entrant dans le bâtiment: si le feu est vert, écrivez votre nom sur le tableau et entrez dans les lieux. Si le feu est rouge, entrez le code, écrivez votre nom sur le tableau et entrez dans les lieux. En sortant du bâtiment: effacez votre nom du tableau. Si le tableau est vide, entrez le code et quittez les lieux. Si le tableau n'est pas vide, juste quittez les lieux.*
>
> *Beim Eintritt ins Gebäude: wenn die Lampe grün ist, schreiben Sie Ihren Namen auf die Tafel und treten Sie in die Räume ein. Wenn die Lampe rot ist, führen Sie die Codezahl ein, schreiben Sie Ihren Namen auf die Tafel und treten Sie in die Räume ein. Beim Ausgang vom Gebäude: beseitigen Sie Ihren Namen von der Tafel. Wenn die Tafel leer ist, führen Sie die Codezahl ein und verlassen Sie die Räume. Wenn die Tafel nicht leer ist, verlassen Sie die Räume nur.*

To illustrate the usefulness of formally specified language in translation, let us look at the result given by a generic machine translation system (available through Alta Vista) from English to German:

> *Wenn das Gebäude betreten wird: wenn das Licht grün ist, schreiben Sie Ihren Namen auf das Brett und betreten Sie die Voraussetzungen. Wenn das Licht rot ist, geben Sie den Code ein, schreiben Sie Ihren Namen auf das Brett und betreten Sie die Voraussetzungen.*
>
> *Wenn das Gebäude gelassen: löschen Sie Ihren Namen vom Brett. Wenn das Brett leer ist, eingeben Sie den Code n und lassen Sie die Voraussetzungen. Wenn das Brett nicht leerer, gerader Urlaub die Voraussetzungen ist.*

(Logicians should be delighted by the translation chosen for *premises*!) In one important respect, the generic system is far superior: it produces a translation for any given input, whereas GF only works for limited grammars. But it seems hopeless to make generic translation systems reliable enough for tasks where great precision is required: for such tasks, translation is much easier to do by specialized systems.

4 The semantics of instructions

The central categories, from the semantic point of view, are **Act** of actions and **Cond** of action conditions. In the category **Instr** of instructions, actions are combined with conditions and other actions to chains of actions. A **state** is semantically a triple

$$S = (A, T, B),$$

where A is a boolean value representing the red/green light, T is the number of names on the whiteboard, and B is the number of persons in the building. The semantics of an action is a function from states to states

$$f : S \rightarrow S,$$

so that the action of entering the code toggles the variable A, and the other actions increase or decrease the variables T and B. The *soundness* of the system is defined in terms of two kinds of *legal states*. The state

$$(\text{True}, 0, 0)$$

is legal, meaning that the alarm is on and that there is no name on the whiteboard, nor anyone in the building. Any state

$$(\text{False}, n, n),$$

where $n \geq 1$, is also legal, meaning that the alarm is off and that the whiteboard has the same number of names as there are people in the building. (Thus it would not be absolutely necessary to write and erase one's own name on the table, but this is a practice with other informational value for the employees.)

A **Text** is a complete instruction, of which two examples were given above, one for entering and one for leaving the building. Like an **Act**, a **Text** is interpreted as a function from states to states. We can now define that

A text is sound, if it maps any given legal state into a legal state.

The soundness of a set of **Text**s can be verified one by one. This is not true of individual actions: after erasing your name and before leaving the building, you are temporarily in an illegal state.

5 Verification

In this section we will give a formal proof of the soundness of the example system. We will proceed roughly as follows:

1. Define the sets of instructions and states (of the system) formally.
2. Define the semantics of the instructions, i.e. define a function of the type $\mathit{Instruction} \rightarrow \mathit{State} \rightarrow \mathit{State}$.
3. Define a predicate P on states where $P(s)$ means that s is "legal".
4. To prove the soundness of an instruction we must prove that it maps legal states to legal states.

We will use Alfa for the verification. This means that we will have a formal representation of the instructions which can be used both in GF (for the translation) and in Alfa (for the proof of correctness).

5.1 Basic Sets

Given the sets *Alarm* (with the two elements **On** and **Off**) and *Counter* (either **Empty** or **OneMore**), we can define the the basic sets of actions and states:

$Act \ \in Set$

$$Act \equiv \textbf{data} \left\{ \begin{array}{l} \textbf{Code} \\ \textbf{Add} \\ \textbf{Erase} \\ \textbf{GoIn} \\ \textbf{GoOut} \\ \textbf{DoNot} \end{array} \right\}$$

$State \ \in Set$

$$State \equiv \textbf{data} \left\{ \ \textbf{State}\,(a \in \ Alarm, nn, pp \in \ Counter) \ \right\}$$

Act represents actions which an employee might be instructed to execute. We represent the state of the system by a triple of the state of the alarm, the number of names on the whiteboard and the number of persons in the building.

Using *Act* and some other simple sets we will eventually define the set *Instr* of instructions. An instruction might be a simple act ("Leave the premises"), but it might also be a sequence of other instructions or a conditional instruction ("If the light is green, enter the code").

5.2 Semantics of Instructions

We would like to have a function $effAct \in Act \to State \to State$, such that e.g. *effAct* **Add** (**Off**, **Empty**, **OneMore Empty**) would return (**Off**, **OneMore Empty**, **OneMore Empty**). However, this function would be partial, e.g. the result of the application *effAct* **GoOut** (**On**, **Empty**, **Empty**) would not be defined. This means that there are states where certain instructions have no proper meaning — if there are no persons in the building, then the instruction "Leave the premises" is pointless. We therefore introduce the *Maybe*-type and define *effAct* to be a function of the type $Act \to State \to Maybe\ State$

We then go on and eventually define a function $effInstrs \in Instrs \to State \to Maybe\ State$.

5.3 Legal States

The soundness of the system is defined in terms of legal states. There are two kinds of legal states:

1. The rest state (**On**, **Empty**, **Empty**), i.e. the alarm is on, the whiteboard is empty and there are no persons in the building.

2. The work states ($\mathbf{Off}, \mathbf{OneMore}\ c, \mathbf{OneMore}\ c$) (any c), i.e. the alarm is off and there are as many names on the whiteboard as there are persons in the building.

This definition of legal states is captured in a predicate $isLegal \in State \rightarrow Set$. Since we are using the *Maybe*-type we will also define *isLegalM*:

$$IsLegalM\ (ms\ \in\ Maybe\, State) \in Set$$

$$IsLegalM\, \mathbf{Nothing} \equiv Zero$$
$$IsLegalM\ (\mathbf{Just}\ a) \equiv IsLegal\, a$$

The set *Zero* is the empty set, which means that "*isLegalM* **Nothing**" is a false proposition.

5.4 Correctness of Instructions

A complete instruction has the form "When entering the building: ..." or "When leaving the building: ...", and will be represented as an element of the set *Text*:

$$Text\ \in Set$$

$$Text \equiv \mathbf{data} \left\{ \begin{array}{l} \mathbf{Enter}\ (i\ \in\ Instrs) \\ \mathbf{Exit}\ (i\ \in\ Instrs) \end{array} \right\}$$

An instruction is correct if it maps legal states to legal states:

$$correctText\ (t\ \in\ Text) \in Set$$

$$correctText\ (\mathbf{Enter}\ i) \equiv (s\ \in\ State) \rightarrow IsLegal\, s \rightarrow IsLegalM\ (effInstrs\, i\, s)$$
$$correctText\ (\mathbf{Exit}\ i) \equiv (s\ \in\ State) \rightarrow Not\ (boardEmpty\, s) \rightarrow IsLegal\, s \rightarrow IsLegalM\ (effInstrs\, i\, s)$$

Note that in the **Exit** case we assume that the whiteboard is not empty. This assumption is valid since when leaving the building, there has to be at least one person in the building (the one who is leaving), which in turn means that the whiteboard cannot be empty.

5.5 The Proof

Now that we have the definitions of instructions and states, the proof of correctness is easy, Alfa does most of the job for us. The first step is to define the two complete instructions from section 2:

$\mathit{whenEntering} \in \mathit{Instrs}$

$$\mathit{whenEntering} \equiv \mathbf{IISeq}\ (\mathbf{IIAtom}\ (\mathbf{Cond}\ (\mathbf{CAtom}\,\mathbf{Green})\ (\mathbf{ASeq}\ (\mathbf{A2Atom}\,\mathbf{Add})\ \mathbf{GoIn})))\ (\mathbf{Cond}\ (\mathbf{CAtom}\,\mathbf{Red})\ (\mathbf{ASeq}\ (\mathbf{A2Seq}\ (\mathbf{A2Atom}\,\mathbf{Code})\ \mathbf{Add})\ \mathbf{GoIn}))$$

$\mathit{whenLeaving} \in \mathit{Instrs}$

$$\mathit{whenLeaving} \equiv \mathbf{IISeq}\,(\mathbf{IISeq}\,(\mathbf{IIAtom}\,(\mathbf{IAtom}\,(\mathbf{AAtom}\,\mathbf{Erase})))\ (\mathbf{Cond}\,(\mathbf{CAtom}\,\mathbf{Empty})\,(\mathbf{ASeq}\,(\mathbf{A2Atom}\,\mathbf{Code})\,\mathbf{GoOut})))\ (\mathbf{Cond}\,(\mathbf{CAtom}\,\mathbf{NonEmp})\,(\mathbf{AJust}\,\mathbf{GoOut}))$$

The proofs are short enough to be listed here. This is the proof for the first instruction, when entering the building:

$\mathit{proof1} \in \mathit{correctText}\ (\mathbf{Enter}\ \mathit{whenEntering})$

$$\mathit{proof1} \equiv \lambda\, s\ \mathit{sIsLegal} \rightarrow$$
$$\mathbf{case}\ s\ \mathbf{of} \left\{ \begin{array}{l} \mathbf{State\,On\,Empty\,Empty} \rightarrow \mathit{tt} \\ \mathbf{State\,On\,Empty}\ (\mathbf{OneMore}\ c) \rightarrow \mathit{absurdElim\ sIsLegal} \\ \mathbf{State\,On}\ (\mathbf{OneMore}\ c)\ \mathbf{Empty} \rightarrow \mathit{absurdElim\ sIsLegal} \\ \mathbf{State\,On}\ (\mathbf{OneMore}\ c)\ (\mathbf{OneMore}\ c') \rightarrow \mathit{absurdElim\ sIsLegal} \\ \mathbf{State\,Off\,Empty\,Empty} \rightarrow \mathit{absurdElim\ sIsLegal} \\ \mathbf{State\,Off\,Empty}\ (\mathbf{OneMore}\ c) \rightarrow \mathit{absurdElim\ sIsLegal} \\ \mathbf{State\,Off}\ (\mathbf{OneMore}\ c)\ \mathbf{Empty} \rightarrow \mathit{absurdElim\ sIsLegal} \\ \mathbf{State\,Off}\ (\mathbf{OneMore}\ c)\ (\mathbf{OneMore}\ c') \rightarrow \mathit{sIsLegal} \end{array} \right\}$$

Most of the cases in the case analysis on s above are illegal states, thus we can use absurd elimination[1] on the proof *sIsLegal*.

The second proof, when leaving the building, is similar (but requires a bit more case analysis):

$\mathit{proof2} \in \mathit{correctText}\ (\mathbf{Exit}\ \mathit{whenLeaving})$

[1]The function *absurdElim* has the type $\mathit{Zero} \rightarrow A$, given that $A \in \mathit{Type}$.

$$
\begin{array}{l}
\mathit{proof2} \equiv \lambda\, s\ \mathit{notEmpty}\ \mathit{sIsLegal} \rightarrow \\
\mathbf{case}\ s\ \mathbf{of} \left\{ \begin{array}{l}
\mathbf{State\ On\ Empty\ Empty} \rightarrow \mathit{absurdElim}\ (\mathit{notEmpty}\, \mathbf{tt}) \\
\mathbf{State\ On\ Empty}\ (\mathbf{OneMore}\ c) \rightarrow \mathit{absurdElim}\ \mathit{sIsLegal} \\
\mathbf{State\ On}\ (\mathbf{OneMore}\ c)\ \mathbf{Empty} \rightarrow \mathit{absurdElim}\ \mathit{sIsLegal} \\
\mathbf{State\ On}\ (\mathbf{OneMore}\ c)\ (\mathbf{OneMore}\ c') \rightarrow \mathit{absurdElim}\ \mathit{sIsLegal} \\
\mathbf{State\ Off\ Empty\ Empty} \rightarrow \mathit{absurdElim}\ \mathit{sIsLegal} \\
\mathbf{State\ Off\ Empty}\ (\mathbf{OneMore}\ c) \rightarrow \mathit{absurdElim}\ \mathit{sIsLegal} \\
\mathbf{State\ Off}\ (\mathbf{OneMore}\ c)\ \mathbf{Empty} \rightarrow \mathit{absurdElim}\ \mathit{sIsLegal} \\
\mathbf{State\ Off}\ (\mathbf{OneMore\ Empty})\ (\mathbf{OneMore\ Empty}) \rightarrow \mathbf{tt} \\
\mathbf{State\ Off}\ (\mathbf{OneMore\ Empty})\ (\mathbf{OneMore}\ (\mathbf{OneMore}\ \mathit{c0})) \rightarrow \\
\quad \mathit{absurdElim}\ \mathit{sIsLegal} \\
\mathbf{State\ Off}\ (\mathbf{OneMore}\ (\mathbf{OneMore}\ \mathit{c0}))\ (\mathbf{OneMore\ Empty}) \rightarrow \\
\quad \mathit{absurdElim}\ \mathit{sIsLegal} \\
\mathbf{State\ Off}\ (\mathbf{OneMore}\ (\mathbf{OneMore}\ \mathit{c0}))\ (\mathbf{OneMore}\ (\mathbf{OneMore}\ \mathit{c1})) \rightarrow \\
\quad \mathit{sIsLegal}
\end{array} \right\}
\end{array}
$$

6 Conclusions and future work

We have argued that the joint use of formal linguistics and program verification can improve the reliability of human-readable instructions, in the same way as program verification alone improves the reliability of machine-readable instructions (=programs). We have illustrated the idea by a small real-world example, for which we have successfully performed a verification as well as automatic translations to different languages.

A natural question is what kinds of documents are amenable to this kind of treatment. For the application of the method to a given domain to be both possible and useful, the following criteria should be fulfilled:

- Texts in the domain can be written in a limited language.
- The domain has a formally definable semantics.
- There is a need for the texts to be reliable.
- There is a need to produce and maintain translations into different languages.

In some applications, the reliability aspect is the most important one, in others, the maintainability: for instance, user manuals are documents that have to be reproduced over and over again in slightly modified versions (as the product itself is modified) and into new market areas. The notion of verification here is not so much that of a complete mathematical proof as that of consistency checks between different parts of the document. Both consistency checks and translations profit from the use of a formalized language, which improves both the quality of the documentation and the "time to market" of the product.

The application of the idea to really large tasks, such as nuclear power-plants or aeronautic industry, does not seem to lie within immediate reach, but some work has been done on bigger examples than ours: Brun & al. [1] have studied a similar method in collaboration with pharmaceutic industry. In their work, instructions for using drugs are generated from a formal source into English and French, and some consistency checks are performed on the source. A syntax editor is used for creating the formal source.

References

[1] C. Brun, M. Dymetman, and V. Lux, Document Structure and Multilingual Authoring, *Proceedings of First International Natural Language Generation Conference (INLG 2000)*, Mitzpe Ramon, Israel, June 2000.

[2] N. Chomsky, *Syntactic Structures*, Mouton, The Hague, 1957.

[3] C. Coquand, Agda Homepage, `http://www.cs.chalmers.se/~catarina/agda/`, Göteborg, 2000.

[4] Th. Hallgren, Alfa Homepage, `http://www.cs.chalmers.se/~hallgren/Alfa/`, Göteborg, 2000.

[5] V. Lux, *Élaboration d'un Français rationalisé étendu modulaire (FREM) pour les manuels de maintenance d'aéronefs*, doctoral dissertation, University of Paris 7, 1998.

[6] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory. An introduction.* Oxford University Press, 1990.

[7] A. Ranta, GF Homepage, `http://www.cs.chalmers.se/~aarne/GF/`, Göteborg, 2000.