VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PRO LIDI SROZUMITELNÝ JAZYK TEMPORÁLNÍ LOGIKY

TEMPORAL LOGIC FOR MAN

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE LUKÁŠ ŽILKA
AUTHOR

VEDOUCÍ PRÁCE Ing. ALEŠ SMRČKA
SUPERVISOR

BRNO 2010

Abstrakt

Tato práce se zabývá automatickým překladem z přirozeného jazyka do temporální logiky. Existující výzkum na toto téma je shrnut a práce je na něm založena. Pro specifikaci temporálních vlastností, je vytvořen kontrolovaný jazyk, podmnožina anglického jazyka. Hlavním přínosem práce jsou algoritmy pro překlad mezi přirozeným jazykem a temporální logikou, založený na zpracovávání a prohledávání vzorů v gramatických závislostech Standfordského parseru angličtiny. Další směr vývoje je diskutován na konci.

Abstract

The work deals with the automated translation of a natural language to temporal logic. Existing research attempts are summarized and built upon. For specificating the temporal properties a subset of English is introduced. The main contribution of the work is the proposed algorithm of translation of a property in the given language to LTL temporal logic, based on processing of and finding patterns in grammatical dependencies of the Stanford English Parser. Future research directions are discussed at the end.

Klíčová slova

temporální logika, LTL, NLP, omezený jazyk, angličtina

Keywords

temporal logic, LTL, NLP, controlled language, English

Citations

Lukáš Žilka: Temporal Logic for Man, Bachelor's thesis, Brno, BUT FIT, 2010

Temporal Logic for Man

Declaration

I declare that this thesis is my own account of my research and contains as its main content work which has not been previously submitted for a degree at any tertiary educational institution.

. . . .
Lukáš Žilka
May 14, 2010

Acknowledgements

Many Thanks to my thesis advisor Ing. Aleš Smrčka for his willingness, support, guidance and priceless advices.

Contents

Chapter 1

Introduction

Hardware and software development advances rapidly, but verification of the developed systems, namely the use of formal methods, lacks wide spread due to its need of an experienced user. With those systems gradually becoming an indispensable part of our lives, everyone can experience the consequences of the development process that suffers from insufficient verification. As a result, the developers of these systems pay more attention to automatic verification and validation methods.

A lot of effort has been put into development of tools supporting the use of formal methods. In general, the verification process consists of preprocessing the systems to be verified (e.g. creating a model of them, or a simulation of an environment), and specifying and verifying their properties. The process of preprocessing the system and its subsequent verification are mature enough to be used in practice thus the last thing that prevents model checking from being widely used is the property definition process. The properties are usually expressed by temporal logic formulas or assertions that are verified on a model of the verified system. Formulas of temporal logic, in particular, offer great power, but also pose a great difficulty for those who build them. The developers need not know all about temporal logic if they want to express how their system is supposed to behave.

This work proposes the design of an *automatic translation system of properties specified in a natural language*, English in particular, to *LTL temporal logic*, and it also comes with a working implementation of this translation system as a prototype.

At the beginning, background of the field of translation from natural language to temporal logic is described. Then, the design and implementation of the translation system is given, and at the end a summary and future work possibilities are discussed.

1.1 Motivation

When a formal verification process is started, for example a verification of a HW system, its developer needs to write temporal formulas to describe its behavior. This can be done, for instance, by describing a timing diagram of the system (Figure 1.1). The developer describes the system in thier natural language, for example English, and produces the following statements that describe the signal transitions in the diagram:

1. If INTR is set, it holds forever.
2. In one cycle, after INTR is asserted, LOCK is written to one.
3. LOCK is set, until INTA is set.

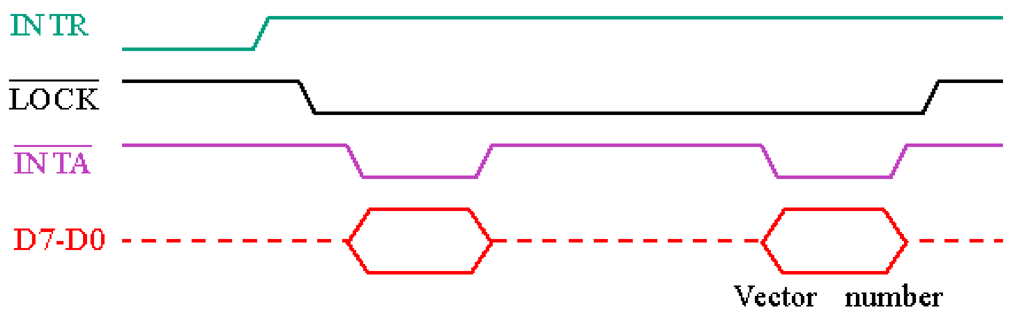


Figure 1.1: Timing diagram of interrupt handling on 8086 processor. Figure reprinted from [4].

4. If LOCK is set, LOCK is eventually deasserted.
5. After DATA is asserted on positive edge of INTA, it is deasserted two cycles later on the negative edge of INTA.

When this is completed, the developer has to rewrite the formulas to a temporal formalism; and this is where the automatic English to LTL translation system can help. The translation system translates the English sentences listed above directly to LTL formulas. It simplifies the process down to inserting the English sentences into the automatic translation tool which produces the following output:

1. `((intr) -> G(intr))`
2. `F((intr) -> X(lock))`
3. `((lock) U (inta))`
4. `((lock) -> F(!lock))`
5. `F((!inta and X inta and data) -> F(inta and X !inta and !data))`

1.2 Current Work

The translation from a natural language to the temporal logic has already been the main topic of several papers and some of them contain interesting information, best of which we tried to use in our research. Although the papers are thorough and high-quality work, their pragmatic use is due to lack of technical documentation unclear and, in general, hard to implement. No working example of an automatic translation system from natural language to temporal logic can be currently found. Usually, the last stage of the translation, which is the actual extraction of temporal and logical relationships, and other information and their processing, are missing.

1.2.1 Natural Language for Hardware Verification: Semantic Interpretation and Model Checking

The work in [8] was created as a part of the PROSPER project (which is very interesting but dead for almost ten years now). It goes through the process of creating a controlled language for the system and behavior specification. We used the information provided to create the limited input language (subset of English) appropriately and to design the mechanisms for its analysis.

1.2.2 Translating from a Natural Language to a Target Temporal Formalism

The author of [5] also deals with the translation from English to temporal logic. He considers several temporal formalism, explains the pros and cons of each, and describes the process of their verification via the Kripke's structure [2]. At the end the author illustrates problems that need to be faced when translating from English to temporal logic, but does not propose any algorithmic solution.

1.2.3 Automatic Translation of Natural Language System Specifications Into Temporal Logic

The work [10] is considering the translation of the natural language to temporal logic through DRS[1]. The author claims to have created a tool that can translate English to temporal logic but its technical documentation or sources are nowhere to be found.

[1]Discourse Representation Structure

Chapter 2

State of the Art

The automatic translation system is based on knowledge from two general areas of computer science, namely *temporal logic* and *natural language processing*. In this chapter, the knowledge from those two fields relevant for this work is discussed.

2.1 Temporal Logic

In this section we will use [2] as a source to briefly introduce the temporal logic. The reader can refer to [2] for more detailed and complete explanation:

Temporal logic is a formalism for describing sequences of transitions between states in a reactive system. In the temporal logics that we will consider, time is not mentioned explicitly; instead, a formula might specify that *eventually* some designated state is reached, or that an error state is *never* entered. Properties like *eventually* or *never* are specified using special *temporal operators*. These operators can also be combined with boolean connectives or nested arbitrarily. Temporal logics differ in the operators that they provide and the semantics of those operators.

2.1.1 The Computation Tree Logic CTL*

Conceptually, CTL* formulas describe properties of *computation tress*. The tree is formed by designating a state in a Kripke structure as the *initial state* and then unwinding the structure into an infinite tree with the designated state at the root, as illustrated in Figure 2.1. The computation tree shows all of the possible executions starting from the initial state.

In CTL* formulas are composed of *path quantifiers* and *temporal operators*. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers $\boldsymbol{A}$ ("for all computation paths") and $\boldsymbol{E}$ ("for some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. There are five basic operators:

- $\boldsymbol{X}$ ("next time") requires that a property holds in the second state of the path.
- The $\boldsymbol{F}$ ("eventually" or "in the future") operator is used to assert that a property will hold at some state on the path.
- $\boldsymbol{G}$ ("always" or "globally") specifies that a property holds at every state on the path.

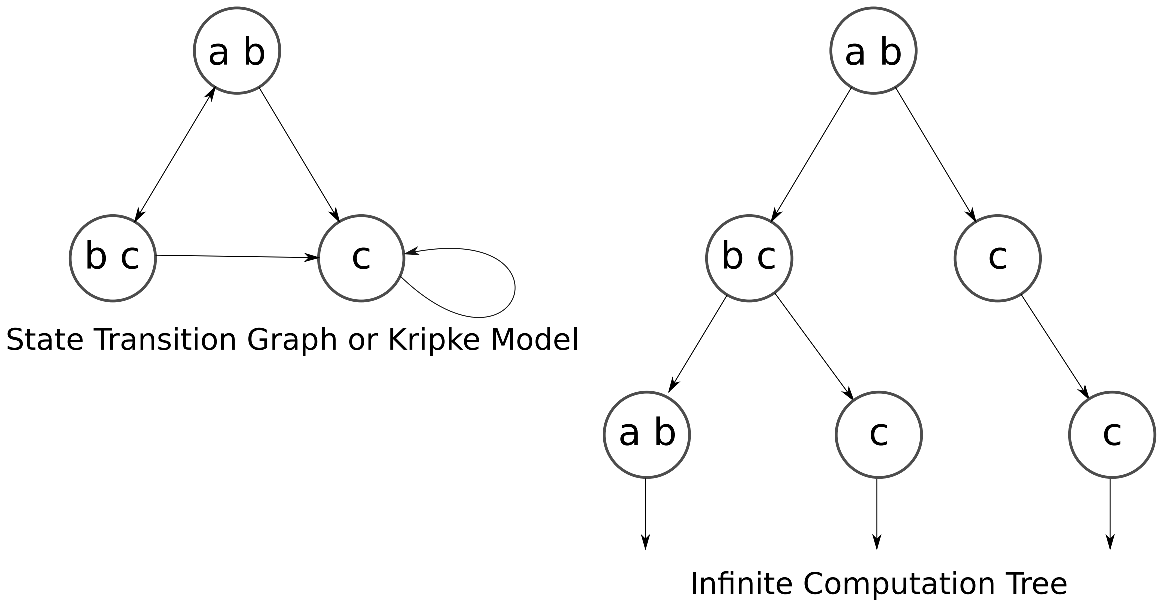

a b
b c
c
State Transition Graph or Kripke Model
a b
b c
c
a b
c
c
Infinite Computation Tree

Figure 2.1: Unwind State Graph to obtain Infinite Tree. Figure reprinted from [2].

- The $\boldsymbol{U}$ ("until") operator is a bit more complicated since it is used to combine two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.
- $\boldsymbol{R}$ ("release") is the logical dual of the $\boldsymbol{U}$ operator. It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

2.1.2 Linear Temporal Logic LTL

Linear Temporal Logic (LTL), consists of formulas that have the form $\boldsymbol{A}f$ where f is a path formula in which the only state subformulas permitted are atomic prepositions. More precisely, an LTL path formula is either:

- If $p \in AP$, then p is a path formula,
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\boldsymbol{X}f$, $\boldsymbol{F}f$, $\boldsymbol{G}f$, $f\boldsymbol{U}g$ and $f\boldsymbol{R}g$ are path formulas.

2.2 Natural Language Processing

Natural language processing (NLP) or computer linguistic is a field of computer science that deals with algorithmic natural (human) language processing and understanding. It has several subdisciplines, each aimed at a particular problem of the whole natural language processing and understanding process. In this work, we are interested in the parts of natural language processing that is concerned with language parsing and dependency extraction.

2.2.1 Natural Language Parser

A natural language parser tries to parse a natural language utterance into a parse tree, that represents the grammatical relationship between the individual words of the input sentence. It is then possible to recognize which word is a subject, which is a predicate, etc. The modern natural language parsers work on the probabilistic principle. At first, they build a probabilistic database of words and their relationships, by training on hand-parsed sentences. Then, when they are presented with a new sentence to parse, they try to produce the most likely parse for it, based on the probabilistic database. Their development was one of the biggest breakthroughs in natural language processing in the 1990s [1].

As described in [6], the first step in the parsing process involves dictionary lookup of successive pairs of grammatically tagged words, to give a number of possible continuations to the current parse. Since this lookup will often not be able to unambiguously distinguish the point at which a grammatical constituent should be closed, the second step of the parsing process will have to insert closures and distinguish between alternative parses. It will generate trees representing the possible alternatives, insert closure points for the constituents, and compute a probability for each parse tree from the probability of each constituent within the tree. It will then be able to select a preferred parse or parses for output. The probability of a grammatical constituent is derived from a bank of manually parsed sentences.

2.2.2 Stanford Parser

Stanford Parser is an implementation of a probabilistic parser in Java by the researchers from the Stanford Natural Language Processing Group. It is a set of Java libraries for NLP that makes together a solid unit capable of processing input text in natural language (English, Chinese, Arabic) and of producing part-of-speech tagged text, context-free phrase structure grammar representation, and a typed dependency representation.

2.3 Specification of the Translation System

Our goal was to design the translation between reasonably constrained natural language and temporal formalism, and implement it. We choose English as the natural language and LTL as the temporal formalism, but decided to provide only a solution for translation from a natural language to LTL. The implemented solution in form of a program should take a specification sentence in English as input, and produce correspondent LTL temporal formula as output. Another aim of this work is to explore the area of automatic translation from a natural language to a temporal logic, try to find solutions for imposed problems, and provide possible direction for further research.

2.3.1 Reversed Translation

Experience from this work can be exploited for designing a reverse-translation system (LTL to a natural language) which will be a subject of future work. Then, the reverse-translation can be fused with this forward-translation system, making together a system that provides the user with the means to check, how the system actually understood the user's sentence. We planned to design and implement such system with a simple LTL parser and plain translation that would map each logic construct to a natural language phrase, to comply with the thesis assignment. Although, we eventually decided against it due to lack of added value of such simple system and insufficient time for building a complex one. We dedicated the time to improving and to higher priority features of the forward-translation system.

2.3.2 Supported Language Constructions

Certain restrictions must be placed upon the range of input language and upon the level of abstraction of the input sentences in order to make designing of the translation feasible. The designed translation should be able to correctly translate descriptive sentences, that give the evidence on how the states in system change. It is not going to support explanatory, and other sentences, that express the system properties in higher terms of abstraction.

The translation should be able to process the following language constructions:

- Simple phrases carrying their meaning in subject and predicate (e.g. *The signal A is set.*)
- phrases describing state transitions of signals
- prepositional clauses (e.g. *After freeing the memory, ...*; *ACK is set on the positive edge of clock*)
- purpose clauses (e.g. *To deassert the signal X, ...*)
- prepositions: while, when, after, before, until, whenever, ...
- conjunctions: and, or
- negation expressed as a negative form of a verb (e.g. *memory is not freed*), or a negation of the subject (e.g. *no memory is freed*)
- formulations expressing temporal properties of global validity (G), future (F), next state(s) (X) in LTL
- multiple subjects/objects
- anaphore (e.g. *The memory can't be used, after it was freed.*) — correctly resolve *"it"* as *"memory"*

Unsupported Language Constructions

- ellipses ("Signal X is asserted on the negative edge of clock and zeroed on the positive.")
- sentences that contain abstract or higher semantical meaning ("Every request on the line REQ must be eventually acknowledged on the line ACK.")
- other, complicated language utterances

Chapter 3

Design of the Translation System

In this chapter, the design of the whole translation process along with all alghoritms and procedures needed for implementation is described. At first, the capabilities and limitations of the input language are defined, then, a detailed description of the parts of the translation process follows.

3.1 Notions

Sentence is an arbitrary sentence in English. A sentence consists of at least one *independent temporal unit*.

Independent temporal unit (or ITU) is a part of sentence (usually the whole sentence), that contains exactly one main phrase and arbitrary number of subordinate phrases. An independent temporal unit consists of at least one *phrase*.

Phrase is a part of sentence that consists of exactly one predicate, one or more subject(s) and arbitrary other parts-of-speech. A phrase contains one main clause and arbitrary number of other clauses.

Clause is a part of phrase, and basic element of expressing the meaning.

3.2 Outline

The proposed translation proceeds in several steps which are discussed in more details later:

- The *input text is processed* (analyzed) with the Stanford Parser and the parsed sentence along with its list of typed dependencies is produced.
- Sentences that could stand on their own and thus do not affect each other *are separated*, and from this point on processed individually (a sentence that can stand on its own is an independent temporal unit).
- *Predicates are extracted* from the sentences.
- *Phrases are extracted* from the sentences.

- *Phrases are classified* according to their function in the sentence.
- *Multiple subjects/objects* are processed.
- Meaning of each phrase is *inferred*.
- Phrases are *put into relation* with each other.
- LTL formula is *synthesized*.

3.3 Language

The English language has been chosen as the input language for several reasons. Mainly due to the fact that it is a language that the software and hardware developers are familiar with as it is the language of science and information technology, and it is also simple enough to facilitate its computer processing. English is also widely known, reasonably easy to learn and its automatic processing has been throughly researched, as a resul of which a great number of tools for its processing already exist and we, therefore, can make use of them.

We have to reduce English to a so called *controlled English* (or limited English), which is a subset of English, to allow for its algorithmic processing. To achieve creation of a proper subset of English that still remains as natural for the developers as possible, we create a corpus of sentences from various specification documents. Then, we use the corpus and so called *hierarchical language design* (Section 3.3.1) procedure as a foundation for designing both the translation process and the controlled English.

3.3.1 Hierarchical Language Design

We use the iterative and hierarchical process for controlled language creation described in [7]. It proceeds in several steps and in each one of them a new and richer language is created.

- The first language is created by literal translation of the constitutive elements of LTL to English (e.g. "The signal WR or the signal RD are not set until the EN signal is set.").
- The second one is created by adding synonymous constructs to the first one (e.g. set = assert, write to one, ...).
- To the third language, aliases are added to cover more complicated patterns (compositional mapping; e.g. "change on the signal A").
- The fourth language contains even abstract language structures that do not necessarily describe behavior of signals but rather represent an abstract pattern of the behavior of signals (e.g. "Every request has to be eventually acknowledged.").

It is sufficient if the translation process can process the sentences from the first language and it has yet the full expressing power of LTL (although, it would be extremely inconvenient for developers to write specification sentences in the first language). The translation proposed in this work is able to process sentences up to the third language.

3.3.2 Controlled English

Sentence Structure Definition

Input sentence (text) consists of at least one independent temporal unit, which consists of at least one phrase.

Independent temporal units in a sentence are joined together with conjunctions "and", and "or", and each independent temporal unit consists of at least one phrase. Phrases are delimited either by conjunctions, by comma (','), or just by meaning. There are 3 types of phrases in sentence:

- *Conditions* express what has to hold in order for the main phrase, that the condition is related to, to be true.
- *Consequences* express when does the main phrase, that the consequence is related to, cease to hold.
- *Main phrases* express the core meaning.

Each phrase always contain a main phrase and optionally other clauses that extend or alter its meaning. The other clauses are always related to the main clause and are inert towards the rest of the sentence:

- *Main clauses* express the core meaning of the sentence (usually by subject and predicate).
- *Prepositional clauses* express condition for or consequence of the main clause (depending on the preposition).
- *Purpose clauses* express that in order for it to be true the meaning of the main clause has to be true.

Dictionary

We have to limit the dictionary of input language. Words are in their stemmed form. The list of words in this dictionary is rather illustrative than exhaustive as it should be easily expandable:

- *signal drop* – deactivate, be down, disable, drop, fall, be false, be low, negate, unset, write to zero, zero,
- *signal rise* – activate, assert, enable, happen, hold, be high, rise, set, be true
- *signal* – line, signal
- *conditions* – as soon as, if, once, since, when, whenever, while
- *temporal determination* – before, until, after
- *temporal succession* – then, afterwards
- *clock cycle* – cycle, tick
- *coordinating conjunctions* – and, or

- *validity modificators* – infinitely often, globally, always, sometimes, in the future, eventually
- *pronouns* – it, former, latter
- *auxiliary verbs* – is, do

Conditions

Conditions appear in form of "$\boldsymbol{\gamma}$ $\boldsymbol{\pi}$", where γ is one of the conditional prepositions (e.g.: after, as, as soon as, if, once, since, when, whenever, while, only while), and π is the rest of the phrase (carrying the rest of the phrase's meaning).

Consequences

Consequences appear in form of "$\boldsymbol{\gamma}$ $\boldsymbol{\pi}$", where γ is one of the consequence prepositions (e.g.: before, until), and π is the rest of the phrase.

Validity modifiers

We need to express the three of the unary LTL operators — future $\boldsymbol{F}$, next time $\boldsymbol{X}$, and global validity $\boldsymbol{G}$.

- $\boldsymbol{F}$ – sometimes, in the future, eventually
- $\boldsymbol{X}$ – in X cycles, X cycles later
- $\boldsymbol{G}$ – infinitely often, globally, always

Negation

English knows two ways of expressing negation of a statement (excluding the case when noun or verb alone convey the negative meaning):

- auxiliary verb + *not* + verb
- *no* + noun

Fillers

Phrase can contain also language constructions that serve linguistic purpose but convey no information about temporal or logical relations (for instance *the, a, can, must, ...*).

Phrase Structure

Each phrase consists of:

- *main clause*, that consists of subject, predicate, optionally conditional and consequential constructions, validity modifiers and filler constructions
- optionally, *prepositional clause*, that consists of object, action, optionally validity modifiers and filler constructions
- optionally, *purpose clause*, that consists of object, action, optionally validity modifiers and filler constructions

3.4 Target Formalism

From many temporal logics that exist today (modal logic, predicate logic, CTL*, CTL, LTL, ...), we chose *LTL*, a sub-logic of *CTL**, as the target formalism, because algorithms for model checking with LTL have reasonable time complexity and are currently very well usable in practice, and also for LTL's similarity to natural language and to people's understanding.

LTL has 5 temporal operators (2.1.2), but we chose to support only 4 of them; we omitted the release temporal operator, because we found that it has no equivalent in natural language expressions.

3.5 Temporal Tree

For the description of the temporal relationship of the individual *temporal elements* of the sentence, we define the *temporal tree*. The whole translation process aims at creating a corresponding temporal tree to the input sentence. Temporal tree is a structure which is designed to be eventually transformed to LTL formula and reflect logical and temporal relationship between phrases and clauses.

3.5.1 Significance

Each node in the temporal tree can either be a terminal node, or a tree node. A terminal node represents an atomic expression (i.e. "X", "freed(memory)"); a tree node represents a node that has at least one child and works as a connector of terminal nodes or other tree nodes. Both the terminal node and the tree node can be marked with tags that signify their temporal or other meaning.

3.5.2 Definition

Temporal tree is a 6-tuple $T = (V, h, E, U, B, W, M)$:

- V is the finite set of vertices.
- $h \in V$ is the head vertex of the tree.
- $E \subseteq V \times V$ is the set of edges.
- U is the finite set of unary temporal operators.
- B is the finite set of binary temporal operators.
- $W \subseteq V \times U$ assigns a set of unary temporal operators to each vertex.
- $T : V \rightarrow (B \cup \{\bot\})$ assigns zero (expressed by $\bot$) or one binary temporal operator to each vertex.
- $M : V \rightarrow \text{meaning\_of}(V)$ assigns the meaning to particular vertices [1].

Each vertex that has more than one child must have exactly one binary temporal operator assigned.

[1]Note that the function `meaning_of` is not important for the definition of the temporal tree, since it does not represent any valuable information for the translation process

3.5.3 Transformation of Temporal Tree into LTL Formula

For LTL are $U \in \{G, F, X\}$ (global validity, future, next state), and $B \in \{U, \rightarrow, \vee, \wedge\}$ (until, implication, or, and). Transformation of the tree into LTL formula is defined in Algorithm 1.

Algorithm 1 Temporal tree synthesis

```
1: procedure tt_to_LTL(T):
2: current_node ← head_of(T)
3: unary_ops ← get_unary_operators_of(T)
4: binary_op ← get_binary_operator_of(T)
5: if binary_op is not ⊥ then
6:     return unary_ops + "(" + tt_to_LTL(first_child) + binary_op + tt_to_LTL(second_child)
       + ")"
7: else
8:     return unary_ops + "(" + get_meaning(T) + ")"
9: end if
```

3.6 Stanford Parser

Stanford parser is used as the preprocessor of the input sentence. It takes a sentence in natural language and marks it with part-of-speech marks, builds tree representation of the sentence from the sentence's context-free-phrase-structure-grammar parse, and eventually builds a list of *typed dependencies*. The list of typed dependencies is the part that is the most interesting for the translation process and is the integral part of its language understanding part.

3.6.1 Typed Dependencies

The Stanford typed dependencies is a way of presenting the information about a sentence's parse in a human-friendly form; it is the opposite to the phrase structure representation which is the source for building the typed dependencies list. Typed dependencies is a list of grammatical relationships between two words in the parsed sentence (such as *subject* relation, *adverbial modifier* relation, ...). The researchers from the Stanford Parser research group say in [3]: "Our experience is that this simple, uniform representation is quite accessible to non-linguists thinking about tasks involving information extraction from text and is quite effective in relation extraction applications."

Typed dependency is a relationship between two words in sentence. The "source" (first) word of the relation is called "head", the "destination" (second) is called "tail".

In this work, we use the following notation of the typed dependencies:
`<type>(<head>,<tail>)`

The list of all typed dependencies included in the Stanford Parser along with their thorough explanations can be found in Stanford typed dependencies manual [3].

Example After parsing sentence "After freeing memory, it cannot be used until it is allocated again." The Stanford Parser produces following typed dependencies (Listing 3.2) which can be visualized as a directed graph (Figure 3.1).

Listing 3.1: Typed dependencies list

```
prepc_after(use-9, free-2)
dobj(free-2, memory-3)
nsubjpass(use-9, it-5)
aux(use-9, can-6)
neg(use-9, not-7)
auxpass(use-9, be-8)
mark(allocate-13, until-10)
nsubjpass(allocate-13, it-11)
auxpass(allocate-13, be-12)
advcl(use-9, allocate-13)
advmod(allocate-13, again-14)
```

Here are explanations of a few important typed dependencies from the Listing 3.2:

- `mark(head, tail)` – connects a subordinating conjunction to the predicate (such as “until”, “when”, ...)
- `advmod(head, tail)` – connects an adverb to the predicate; it modifies its meaning
- `{subj, nsubj, nsubjpass, xsubj, csubj, csubjpass}(head, tail)` – connects a subject (different types of subjects) to the predicate
- `neg(head, tail)` – denotes negation of its head word (usually the predicate)

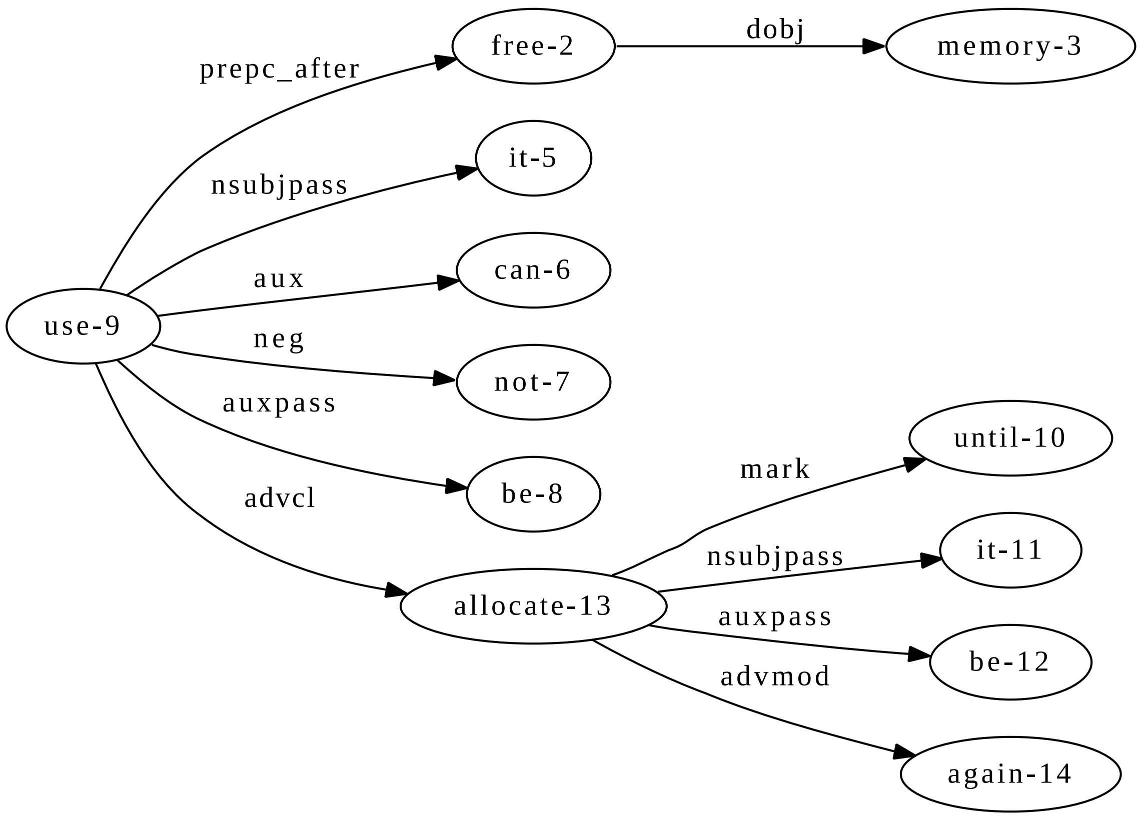


Figure 3.1: Typed dependencies graph

3.7 Sentence Separation

A raw input sentence needs to be separated in order to isolate the independent temporal units contained in it. It is important to isolate them because they can each be processed on its own, as none of them has any influence on the temporal course described by another independent unit, and yet they are part of one sentence. Each independent temporal unit of the sentence carries its own temporal meaning and all of the units can be joined together at the end of the translation process with appropriate logic constructs "and", or "or".

3.7.1 Algorithm of Separation of Independent Temporal Units

Algorithm 2 Independent temporal unit separation algorithm

```
Require: set of typed dependencies (deps)
Ensure: set of typed dependencies for each independent temporal unit (result)
  result ← [] //list of ITUs
  remove all conj dependencies that join predicates
  while len(deps) > 0 do
    currentITU ← [] // for current independent temporal unit
    currentITU.append(pop(deps))
    while exist(dependency from/to a word from currentITU in deps) do
      currentITU.append(pop(deps, dependency from/to a word from currentITU))
    end while
  end while
```

3.8 Predicate Extraction

As each predicate has at least one subject, the algorithm for predicate extraction is straightforward. In the list of typed dependencies, there is a special dependency called `subj` which says that the word in the tail of the dependency (relationship) is the subject of the word in the head of the dependency. By going through all of the `subj` dependencies it is possible to find all predicates of the whole sentence.

The algorithm for extraction searches for typed dependencies of type `subj` or inherited[2], and looks at the head of the relationship which is always the predicate of the sentence.

[2]There is inheritance in types of the typed dependencies of the Stanford Parser.

3.8.1 Predicate Extraction Algorithm

Algorithm 3 Predicate extraction algorithm

```
Require: set of typed dependencies (deps)
Ensure: set of predicates
  result ← [] //list of predicates
  for all td in typed dependencies do
    if td is subj relation then
      result.append( td )
    end if
  end for
```

3.9 Phrase Separation

Phrase separation is very similar to Sentence separation described in Section 3.7. Phrases in each independent temporal unit have to be isolated, so that they can be analyzed and classified separately. The core of each phrase is predicate and subject. Algorithm for the predicate extraction was presented in Section 3.8.

Phrase separation does not exactly proceed how would one expect a separation algorithm to, but it rather identifies each phrase by evolving and unwrapping from its predicate. It first finds a list of predicates as presented in Section 3.8, and then, for each predicate it searches for all typed dependencies that are transitively related to it except for the dependencies that represent relationship to other predicates. As a result, groups of typed dependencies that are directly or transitively related to each predicate are created.

3.9.1 Phrase Separation Algorithm

Algorithm 4 Phrase separation algorithm

```
Require: set of typed dependencies
Ensure: set of typed dependencies for each phrase
  result ← [] //set of sets of typed dependencies
  predicates ← extractPredicates(deps)
  for all pred in predicates do
    currPredDeps ← []
    while dependency from/to pred exists in deps do
      add it to currPredDeps
      remove it from deps
    end while
  end for
```

3.10 Phrase Classification

For further processing, each phrase needs to be classified as either *main phrase*, *precondition phrase*, or *consequence phrase*. Both *precondition phrase* and *consequence phrase* express what has to be true, or will have to be true before, or after the main phrase is true. The classification alone is accomplished by looking at the presence of particular dependencies in the typed dependencies list of each phrase.

The classification provides means for synthetization alghoritms to create a rough temporal structure of the sentence, that can be further refined by the meaning inference processes.

There are many ways of expressing preconditions, or consequences in English, so we picked only the common ones as the base and count on flexible implementation for extension possibility. They are described in the Section 3.3 and further discussed in the following paragraphs.

Precondition Phrase Precondition phrases are distinctive by presence of particular conjunctions as: if, when, whenever, while, after. They are related to predicate of the sentence by the following typed dependencies:

- *if, after*: `mark(<predicate>, if),mark(<predicate>, after)`
- *when, whenever*: `advmod(<predicate>, when),advmod(<predicate>, whenever)`

If one of those dependencies is present in the classified phrase, it is marked as a precondition phrase.

Consequence phrase Subsequence phrases are distinctive by presence of particular conjunctions as: until, before. They are related to predicate of the sentence by the following typed dependencies:

- *until, before*: `mark(<predicate>, until), mark(<predicate>, before)`

So if one of those dependencies is present in the classified phrase it is marked as a consequence phrase.

3.11 Multiple Subjects/Objects

Each subject/object is identified by either `subj` or `obj` relation in typed dependencies, but only the first subject/object is related this way. If multiple subjects are present in the sentence, the additional subjects/objects are related to the first (or subsequent) subject/object by `conj` relation; the relations create a linear list of all of the subjects/objects linked by either "and", or "or" conjunction. The procedures for meaning inference has to count with the possible multiplicity of subjects/objects.

Listing 3.2: Typed dependencies with multiple subjects

```
nsubjpass(set-7, EN-2)
conj_and(EN-2, WR-5)
conj_and(WR-5, CLK-7)
```

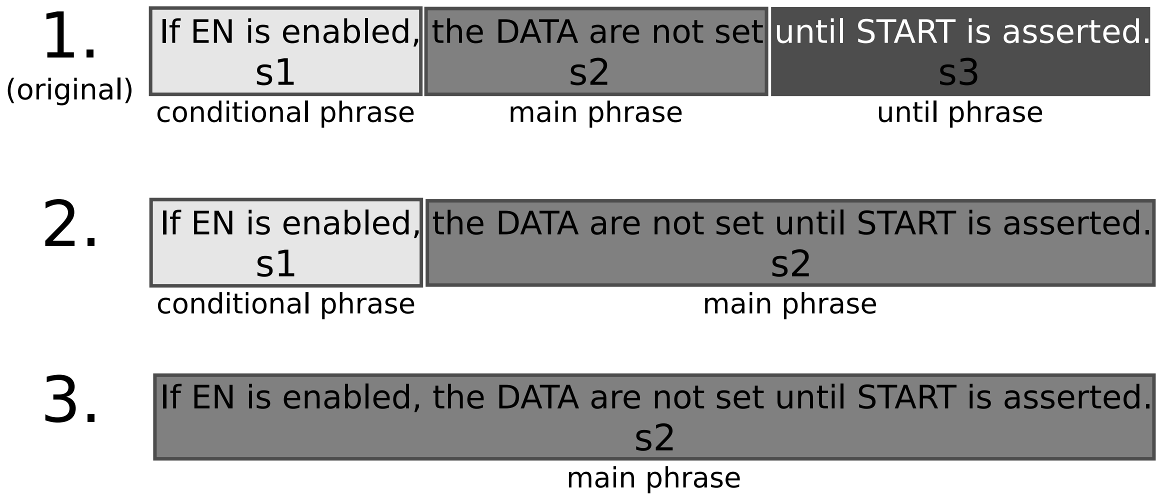


Figure 3.2: Phrase reduction order

3.12 Temporal Relationship Between Phrases

Temporal relationship of phrases in sentence is given partly by their content and partly by their position in the sentence; extraction of the temporal-relationship related content is described above in Section 3.10, and rules for inferring the temporal relationship that consider both the phrase's position in the sentence and the content information are presented in this section.

Each independent temporal unit (sentence in this section for clarity) contains one main phrase that carries the central meaning which is modified by the rest of the phrases (modifying phrases in this section for clarity). People tend to put modifying phrases that stand in the sentence in front of the main phrase in chronological order, and modifying phrases that stand behind the sentence in reversed chronological order. Therefore, when reduction in order to construct a temporal tree is performed, the modifying phrases from behind of the sentence must be reduced at first and the phrases in front of the main phrase later. This is demonstrated in Figure 3.2.

3.13 Meaning Inference

Meaning inference is the last step of sentence analysis. Even though the computer is able to parse the sentence, mark correctly parts-of-speech, and produce list of typed dependencies, all of the words are meaningless from the computer's point of view. There are attempts to bring understanding of individual words closer to computers through various ontology systems, but they are just currently being researched.

Meaning inference of a phrase consists of main, prepositional and purpose clause identification and understanding. Each of the phases involve predicate understanding, subject/object detection and temporal properties extraction from other parts-of-speech of the particular clause.

3.13.1 Clause Identification and Understanding

Main Clause

Consists of subject, predicate and optionally of parts-of-speech related to them.

Prepositional Clause

Consists of a root of the prepositional clause which is usually a gerund, object and optionally of parts-of-speech related to them. Prepositional clause is connected to the predicate of main clause through `prepc` relation with attribute carrying correspondent preposition.

For purposes of understanding the contents of the sentence, the root of the prepositional clause can be considered to be predicate and object to be subject.

Purpose Clause

Consists of a root of the purpose clause which is usually a verb infinitive, object and optionally of parts-of-speech related to them. Purpose clause is connected to the predicate of main clause through `dep` relation, and contain `aux` relation with "to" in its tail, identifying that this is a purpose clause.

For purposes of understanding the contents of the sentence, the root of the purpose clause can be considered to be predicate, and object to be subject.

3.13.2 Subject/Predicate Understanding

Subject and predicate express what is being talked about (subject) and what happens (predicate). From the point of the translation process, predicates need to be analyzed for both, their effect on the rest of the sentence and their function in the phrase. In this work, we aimed at analyzing of predicates that describe behavior of signal in hardware systems, and we left analysis of predicates with complex meaning to future work. Phrases, where the meaning of subject/predicate cannot be inferred, are produced in form of `<predicate>(<subject>)`, otherwise in form defined for the particular translation (the name of the signal in our case).

Setting the signal is expressed by a range of verbs: set, assert, set to one, ... When the predicate is from this group, the meaning of the phrase is expressed by its subject because the subject is the name of the signal that is being talked about, and also because the predicate means that the signal is in positive state which is commonly expressed by its name.

Unsetting the signal is expressed by a range of verbs: unset, zero, ... When the predicate is from this group, the meaning of the phrase is expressed by negation of its subject because the subject is the name of the signal that is being talked about, and predicate from this group conveys negative state of this signal (subject).

3.13.3 Temporal Properties Extraction

Temporal properties are communicated by various parts-of-speech. Here we aim on the basic ones that convey some of the three unary temporal operators in LTL. They are recognized by searching for certain patterns in the list of temporal dependencies.

Global validity English expresses global validity temporal property by adverbs, such as "globally", "always", "from now on", "never", "forever", ... By examining typed dependencies of a sentence with those global validity adverbs, it can be revealed that they are joined to the predicate as follows:

- *never, forever, always*: `advmod(<predicate>, never)`, `advmod(<predicate>, forever)`,...
- *from now on*: `prep(<predicate>, from)`, `pobj(from, now)`, `prep(<predicate>, on)` [3]

If some of the dependencies above are present in the list of dependencies for a particular phrase, that particular phrase conveys global validity, and must be marked with corresponding temporal operator — ***G*** in this case.

Next time English expresses *next time* temporal property by clauses, such as as "in the next state", "X cycles later". Temporal dependencies of a phrase with those clauses contain following items:

- *in the next state*: `prep_in(<predicate>, state)`, and `num(state, next)`
- *X cycles later*: `advmod(<predicate>, later)`, and `dep(later, cycles)`, and `num(cycle, X)` [3]

As can be seen above, the number of states to the future (or past) is expressed by the tail argument of the `num` dependency. That means, the phrase must be marked with the given number of temporal operators for next state, or previous state in case of past. Temporal operator for previous state does not exist in LTL, so it has to be transfered to other phrase if such exist (if it does not, the whole sentence is untranslatable) before the resulting formula is presented to user.

Future English expresses the temporal property of validity at some point in future by phrases, such as "eventually", "in the future", "sometimes", ... By examining typed dependencies of a sentence with the aforementioned phrases, it is revealed that they are joined to the predicate as follows:

[3]

- `pobj(head, tail)` – connects the object of the preposition (tail) to the preposition (head)
- `prep(head, tail)` – connects the preposition (tail) to the predicate (head)
- `prep_in(head, tail)` – connects the tail to the predicate (head) and says they are joined by the preposition *in*
- `num(head, tail)` – connects the number (tail) to the head

- eventually, sometimes — `advmod(<predicate>, eventually),advmod(<predicate>, sometimes)`
- in the future — `prep(<predicate>, in-6),pobj(in, future)`

If some of the dependencies above are present in the list of dependencies for a particular phrase, that particular phrase conveys validity at some point in the future, and must be marked with corresponding temporal operator — *F* in this case.

3.14 Anaphora Resolution

Definition of *anaphora* from [11]: "The use of a linguistic unit, such as a pronoun, to refer back to another unit, as the use of her to refer to Anne in the sentence Anne asked Edward to pass her the salt."

As anaphoras are common in all forms of natural language utterances, including system property specification texts, they need to be dealt with. But the algorithms that has been developed are complicated and the trade-off between a complicated algorithm and a simple approach is negligible in the case of system specification texts, so we opted for the simple one.

If a pronoun is detected in a sentence instead of a noun subject, the anaphora resolution algorithm is performed. It works with the temporal tree representation of the sentence (Algorithm 5).

Algorithm 5 Anaphora resolution algorithm

```
Require: temporal tree (tt), node carrying the phrase with the pronoun (phrase)
Ensure: subject that the pronoun in (phrase) reffers to
  go up the temporal tree, until a node where the node that we ascend from is its second child
  look into the phrase of the first node for subject
```

3.15 Temporal Formula Synthesis

At the final stage, when the temporal tree is built, the final temporal formula is created out of the temporal tree with Algorithm 1.

Chapter 4

Implementation of the Translation System

For implementation, we choose Python for its modularity and widespread availability. It also, along with its meta-programming features, offers building of an elegant, extensible and lightweight solution that does not loose manageability, without major speed decrease.

Because the Stanford Parser, one of the integral parts of the whole system, is written in Java, we use the JPype Python library which facilitates integration of Java programs and libraries into Python, to work with the Standford Parser.

In this chapter, we first look at the system as a whole, present individual parts, and the main program flow. Subsequently, the configuration of the translation system is described; the principles of the phrase patterns that serve as the main sentence analysis tool of the translation system are described; a list of tags and their explanations is given; and, at the end, the internals of the final synthesis of the temporal formula are described.

4.1 JPype

JPype is a Python library that enables a Python programmer to interface with Java libraries. It joins both Virtual Machines (Python and JVM) at the native level, and provides a fast and lightweight interface to re-use existing Java libraries in Python programs [9].

4.2 Modules of the Translation System

The program is divided into following modules:

- `main` — the runnable part of the program; loads configuration, parses the command-line options, creates the objects for translation, runs the translation and presents the result to the user
- `parser` — interface to the Stanford Parser
- `ttree` — class that represents the temporal tree
- `groups` — classes that load named groups of words from file
- `langtools` — classes that work linguistically with the typed dependencies; sentence separation, phrase extraction, predicate, subject and object extraction

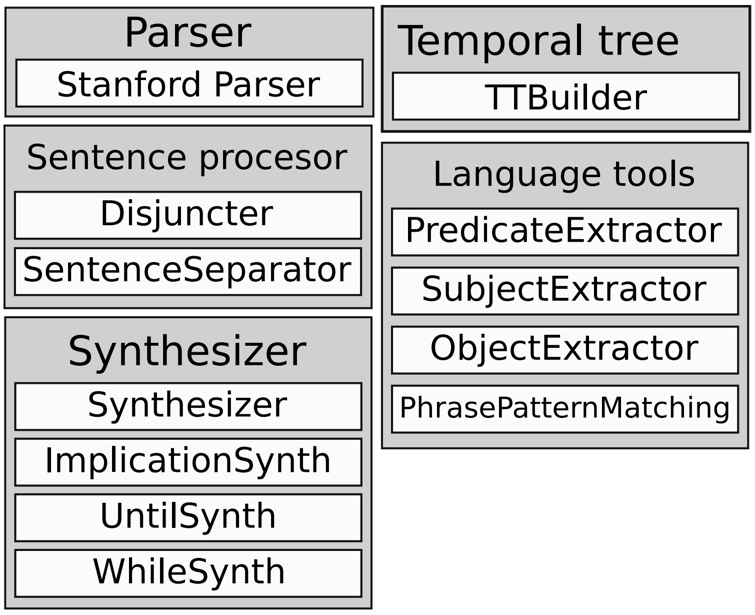


Figure 4.1: The program's architecture. Gray boxes depict the important modules of the translation system, and the white boxes depict the important classes that implements the main functionality of the translation system.

- `patterns` — classes for assigning tags to phrases by matching language patterns with the typed dependencies
- `synth` — classes providing the final synthesis of the temporal tree to LTL formula
- `translator` — class that hooks up all of the modules of the system; the designated entry point of the translation program

A brief view of the program's architecture is depicted in the Figure 4.1.

4.3 Program Flow

The program proceeds as follows:

1. The JVM and the Stanford Parser are loaded in.
2. The Standford Parser parses the sentence and creates the typed dependencies for it.
3. The `Disjuncter` separates the typed dependencies set into sets of typed dependencies, each representing one independent temporal unit of the original sentence.
4. The `SentenceSplitter` creates a set of phrases out of each independent temporal unit.
5. Each phrase it tagged by the `PhrasePatternDirectory` routines.
6. The `TTBuilder` (Temporal Tree Builder) is engaged to synthesize the tagged phrases into a LTL formula.

7. LTL formula for each independent temporal unit is printed.

4.4 Configuration of the Translation System

The main configuration of the program is a standard Python file `config.py` which is loaded at startup; other configuration files, as the phrase pattern directory, and synonym groups, are loaded from YAML files (`phrase_patterns.yaml`, `synonym_groups.yaml`). The YAML configuration format was chosen for its lucidity, and easy and comfort editing by humans.

4.4.1 phrase_patterns.yaml

Patterns

The file contains a list of *patterns*. Each *pattern* contains:

- `cls` – tags that are assigned when there is a match between the typed dependencies in a currently tested phrase and the list of typed dependency patterns
- `dependency_list` — a list of *typed dependency patterns*

Typed Dependency Patterns

Each *typed dependency pattern* contains:

- `reln` – relation name expression
- `gov` – head of a relation expression
- `dep` – tail of a relation expression

Expressions

An expression consists of:

- `type` – type of the expression can be `null`, `value`, `group`, `tagarg`, `variable`, `predicate`, `subject`) and additional parameters vary according to the type

Types of expressions (principles of each of the following are explained in the Section 4.5.1):

- `null` – no additional parameters
- `value` – additional parameter `value` with words or a list of words
- `group` – additional parameter `name` with name of the synonym group
- `predicate` – no additional parameter group
- `subject` – no additional parameter group
- `group` – additional parameter `name` with name for/of the variable

Example of the Patterns Configuration

```
# SIGNAL UNSET PATTERNS
- cls: signal_unset
  dependency_list:
  - gov: { type: predicate, group: signal_unset }

- cls: signal_unset
  dependency_list:
  - gov: { type: predicate, value: write }
    dep: { type: group, name: signal_negative_state }
    reln: { type: value, value: [ prep_to, dobj ] }

- cls: signal_unset
  dependency_list:
  - gov: { type: predicate, value: write }
    dep: { type: value, value: logic }
    reln: { type: value, value: [ prep_to, dobj ] }
  - gov: { type: value, value: logic }
    dep: { type: group, name: signal_negative_state }
```

4.4.2 synonym_groups.yaml

This file contains synonym groups (named groups of words) that are utilized in pattern matching.

It consists of keys that each contain a list of words, with the following syntax:

```
<name of the˜group>: [<word1>, <word2>, ...]
```

Example of the Synonymes Configuration

```
signal_set: [ set, assert, high, enable, activate ]
signal_unset: [ unset, zero, low, disable, deassert, deactivate ]
```

4.5 Phrase Patterns

From the design of the translation it is apparent that the whole analysis process relies on comparing the typed dependencies of the input sentence to certain patterns. Therefore, we aimed to create a flexible pattern recognition system wherein lies the core of the meaning inference of the whole program.

List of typed dependencies for each input sentence is broken down, first, to a set of typed dependencies lists for each independent temporal unit, second, to a set of typed dependencies for each phrase by algorithms described in the chapter 3. Typed dependencies of each phrase are then analyzed by the pattern matching system that checks the typed dependencies of a phrase against given patterns (specified in a configuration file). If a pattern matches, the phrase is decorated with the tags of the pattern, and they stick with it throughout the rest of the translation process. The tags are used further, by the synthesizing part of the translation process.

A pattern has two components—a set of tags, and a set of typed dependency patterns. Tag is a short string that optionally contains an argument. Typed dependency pattern is described in the following section.

4.5.1 Typed dependency patterns

Typed dependency pattern consists of three variables which specify what can be present at which place of a dependency that is being checked, for the comparison to terminate with success:

- dependency name
- head of the dependency
- tail of the dependency

When a typed dependency is being checked against a typed dependency pattern, each of the constituents of the dependency are checked with the pattern separately (name, head, tail). If matching of all three succeeds, the check is successful.

Value of each of those variables (constituents) of a typed dependency pattern can be either:

- *null value* – does not matter, matches all values (= wildcard)
- *a word* – a single word value, that matches only if the tested word is exactly the same
- *a list of words* – a list of words; matches if at least one of the words from the list exactly matches the tested word
- *a group* – same as a list of words, but the list is retrieved from an external file with synonym groups
- *predicate* – matches only if the tested word is the predicate of the currently tested phrase
- *subject* – matches only if the tested word is the subject (or one of the subjects) of the currently tested phrase
- *variable* – when the matching algorithm sees this dependency pattern for the first time for particular phrase, it saves the word that was matched, under some name to the context and returns success;
  then, later, when matching continues to next typed dependency pattern of the same pattern, if it encounters variable of the same name again, it does not save it to the context as before, but rather looks up the previously saved value from the context and checks if it matches with the tested word;
  this facilitates creating a non-trivial dependency checking
- *tag argument* – passes the tested word to the argument of currently assigned tag (if other constituents of the pattern match)

4.6 Tags

Pattern matching procedures assign various tags to phrases that synthesizers use to synthesize the temporal tree out of the phrases, and interpret their meaning. The following tags are currently supported:

1. Functional:
   - `if` – denotes that this phrase is a head of a condition
   - `until` – denotes that this phrase is a head of an until condition
   - `while` – denotes that this phrase is a head of a while condition
2. Operators:
   - `not` – denotes that this phrase's predicate is negated
   - `future` – denotes that this phrase expresses the future property of LTL
   - `globally` – denotes that this phrase expresses the global validity property of LTL
   - `past` – denotes that this phrase happens relatively in past to the other phrases
   - `cycle (arg)` – denotes that this phrase expresses the next state property of LTL; can have an argument with a number
3. Clause:
   - `prepc` – denotes that a prepositional clause is present in the phrase
   - `purpose_clause` – denotes that a purpose clause is present in the phrase
4. Meaning:
   - `signal_set` – denotes that this phrase expresses that the signal is in positive state (one)
   - `signal_unset` – denotes that this phrase expresses that the signal is in negative state (zero)
   - `edge (arg)` – denotes that this phrase expresses that the signal is just being changed; can have an argument with the direction of the change

4.7 Synthesis

The synthesis is a method for rewriting the data structures created by the information extraction processes into a LTL formula. The extracted information are represented by a set of tags with arguments, assigned to particular phrases. At first, the temporal tree is built out of the phrase list with assigned tags, then, the tree is rewritten to LTL formula.

4.7.1 Building the Temporal Tree

The linear structure retrieved by processing the input sentence by separating and tagging algorithms needs to be transformed to a temporal tree that actually represents temporal relations of the input sentence; the structure consists of a list of phrases with tags. Principles of reduction of the list of phrases were described in Section 3.12 and are used to build the temporal tree by the bottom-up approach (Algorithm 6).

Algorithm 6 Building temporal tree

```
Require: list of tagged phrases (phrases)
Ensure: temporal tree (tt)
 1: tt = Node(main phrase)
 2: while len(phrases) > 0 do
 3:   find main phrase
 4:   if there is a phrase behind main phrase then
 5:     actphrase = first phrase behind main phrase
 6:   else
 7:     actphrase = last phrase before main phrase
 8:   end if
 9:   if class(actphrase) is condition then
10:     tt = Node( 'condition', the phrase, tt )
11:   else
12:     if class(actphrase) is consequence then
13:       tt = Node( 'consequence', the phrase, tt )
14:     else
15:       error, unknown class; the temporal tree cannot be built
16:     end if
17:   end if
18: end while
```

4.7.2 Building the LTL Formula

Building the LTL formula proceeds from the top of the tree down to the bottom. On the path down each node of the tree is visited and asked to rewrite itself to string; each node has its own method of rewriting itself to string (and of including its children), and each node also knows what temporal operators it contains and how to rewrite them to the resulting LTL formula (Algorithm 7).

Algorithm 7 Temporal operators synthesis

```
 1: if current node contains tag "cycle" then
 2:   retrieve the number of cycles from the tag and put corresponding number of the "X"
      operators on the output
 3: else
 4:   if current node contains tag "future" then
 5:     put the "F" operator on the output
 6:   else
 7:     if sibling node contains tag "past" and "im" (immediate) then
 8:       put the "X" operator on the output
 9:     else
10:       put the "F" operator on the output
11:     end if
12:   end if
13: end if
```

Formula Building Rules

When a tree node is asked to rewrite itself to temporal formula it proceeds according to the following rules (each rule is in form of: `<node type>: <rewriting rule>`):

- `TTImplNode: <first child> -> <second child>`
- `TTUntilNode: <first child> U <second child>`
- `TTAndNode: <first child> and <second child>`
- `TTOrNode: <first child> or <second child>`

When a terminal node is asked to rewrite itself to temporal formula, it resolves anaphora, and then rewrites itself to form of: `<operators><predicate>(<subject>)` or `<operators><signal name>`, according to whether the node is of general or signal specific meaning.

Chapter 5

Conclusion and Future Work

We successfully designed and implemented a system that automaticaly translates sentences given in English to LTL formulas. The system is the most suitable for a translation of sentences describing signals behavior in hardware systems, but it is also usable to translate behavior description of other systems. Out of 10 sentences that we aimed to support at the beginning, it is able to translate 8 flawlessly. Problem with the rest is a higher semantical meaning, processing of which we were not yet able to incorporate.

The implemented translation system can be used by software and hardware developers that are considering the idea of formal verification but do not want to dive into temporal logic. After concise familiarization with the system and with the limitations of the input language, developers should be able to simply produce LTL formulas from their English utterances about the system under test. Also, the system could be used in formal verification courses to give the students something they can experiment with when learning about temporal formalisms.

We see the continuation of this work in integration of processing of more abstract language, and extending the range of supported language construct to maximum. This work also offers itself to be extended with advanced NLP text classification methods but it involves collecting a huge number of examples of all variety. When such bank of sentences is built and properly annotated, a number of interesting statistical and machine learning methods could be tried. Another possibility for an extension is an ontology system engagement which after our shallow investigation, seems very promising and could later aid or fully replace the meaning inference parts of this system.

Bibliography

[1] The Stanford Parser: A Statistical Parser. Java Library. [online] http://nlp.stanford.edu/software/lex-parser.shtml.

[2] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. ISBN 0262032708.

[3] M.C. de Marneffe and C.D. Manning. Stanford Typed Dependencies Manual. [online] http://nlp.stanford.edu/software/dependencies_manual.pdf, September 2008.

[4] 8086/88 Device Specifications. [online] http://www.ece.unm.edu/~jimp/310/slides/8086_interrupts.html, April 2010.

[5] R. Fazaeli. Translating from a Natural Language to a Target Temporal Formalism. Final Year Project, Trinity College Dublin, May 2002.

[6] R. Garside and F. Leech. A Probabilistic Parser. In *Proceedings of the second conference on European chapter of the Association for Computational Linguistics*, pages 166–170, Morristown, NJ, USA, 1985. Association for Computational Linguistics. DOI 10.3115/976931.976955.

[7] C. Grover, A. Holt, E. Klein, and M. Moens. Designing a Controlled Language for Interactive Model Checking. In *Proceedings of the Third International Workshop on Controlled Language Applications*, 2000.

[8] A. Holt, E. Klein, and C. Grover. Natural Language for Hardware Verification: Semantic Interpretation and Model Checking. In *Proc. ICoS-1: Inference in Computational Semantics, Amsterdam*. University of Amsterdam, 1999.

[9] JPype: Bridging the Worlds of Java and Python. [online] http://jpype.sourceforge.net/, April 2010.

[10] R. Nelken and N. Francez. Automatic Translation of Natural Language System Specifications Into Temporal Logic. In *8th International Computer Aided Verification Conference*, 1996. DOI 10.1007/3-540-61474-5_83.

[11] J.M. Sinclair, G.A. Wilkes, and W.A. Krebs. *Collins English Dictionary*. HarperCollins, 2003. ISBN 0007232306.

Appendix A

CD Contents

- `README` – system requirements, installation instructions and basic usage
- `src/` – the translation tool with source codes
- `libs/` – current versions of supportive libraries

Appendix B

Corpus of the System Specification Sentences

- The CSn, BSn, OE and AS signals are synchronous to the falling edge of CLKOUT.
- Changes on these signals are triggered by the falling edge of CLKOUT.
- Microcontroller puts /WR signal low wile A/D is low as well and starts bus access.
- One access is completed after two cycles on the parallel bus.
- First bus cycle is always a write cycle and sets the 8bit address.
- Second cycle can be either read or write, depending on state of /WR and /RD lines.
- If PORTxn is written logic one when the pin is configured as an input pin, the pull-up resistor is activated.
- To switch the pull-up resistor off, PORTxn has to be written logic zero or the pin has to be configured as an output pin.
- The bit TOV0 is set when an overflow occurs.

Appendix C

Tutorial of a Simple Usage of the Translation Tool

Introduction

Temporal Translator is a tool that translates sentences from English to LTL formulas.

In this tutorial we will look how to use Temporal Translator to translate English sentences to LTL and we will learn how to extend the range of language that it is able to process.

C.1 Installation

C.1.1 System Requirements

- `Python >=2.5`
- `JPype`
- `Java >=1.6`

C.1.2 Configuration

In order for the translation to work, we need to configure path to Java Virtual Machine, and The Standford Parser PCFG file in `config.py`.

Listing C.1: Example `config.py`

```
java_bin = "/usr/lib/jvm/java-6-sun-1.6.0.16/jre/lib/i386/client/libjvm.so"
parser_pcfg = "englishPCFG.ser"
```

C.2 Command Line Interface

```
tt.py {[-f filename],[sentence]*}
```

- `-f filename` - loads sentences from *filename*; it expects one sentence per line

- `sentence` - the sentence that we want to translate to LTL; any number of input sentences is possible (we are only limited by the system's command-line maximal command length)

Basic example of use

```
$ tt.py ”If EN is set, after the memory was freed, the memory cannot be used,
until it is allocated.”
...
>>> TT was successfully built.
((set(EN)) -> ((freed(memory)) -> F(!(used(memory)) U (allocated(memory)))))
$
```

C.3 Input Language

Input sentences should be descriptive sentences that we use to describe behavior of a program/signal. (E.g. After freeing memory, it cannot be used.)

The program can understand:

- simple phrases carrying their meaning in subject and predicate (e.g. The signal A is set.)
- prepositional clauses (e.g. After freeing the memory, ...; ACK is set on the positive edge of clock)
- prepositions like: while, when, after, before, until, whenever, ...
- negation expressed as a negative form of a verb (e.g. memory is not freed), or a negation of the subject (e.g. no memory is freed)
- temporal modifiers of global validity (G in LTL), future (F), next state(s) (X)
- multiple subjects/objects
- anaphore (e.g. The memory can't be used, after it was freed.) — correctly resolves *it* as *memory*

We can broaden the range of the input language by editing the *pattern directory*, or extending the program itself, but we need to get acquainted with the internals of the program first.

C.4 Extending

In this section, we will look at how to teach the translator new stuff.

C.4.1 Creating a New Rule

Suppose that we want the translator to be able to translate the following sentence into LTL correctly:

```
Immediately after the memory is freed, it cannot be used again.
```

Translator with the current default version of pattern phrase directory can translate this sentence correctly, but suppose it translates the sentence as follows:

```
(F(freed(memory)) -> X(!used(memory)))
```

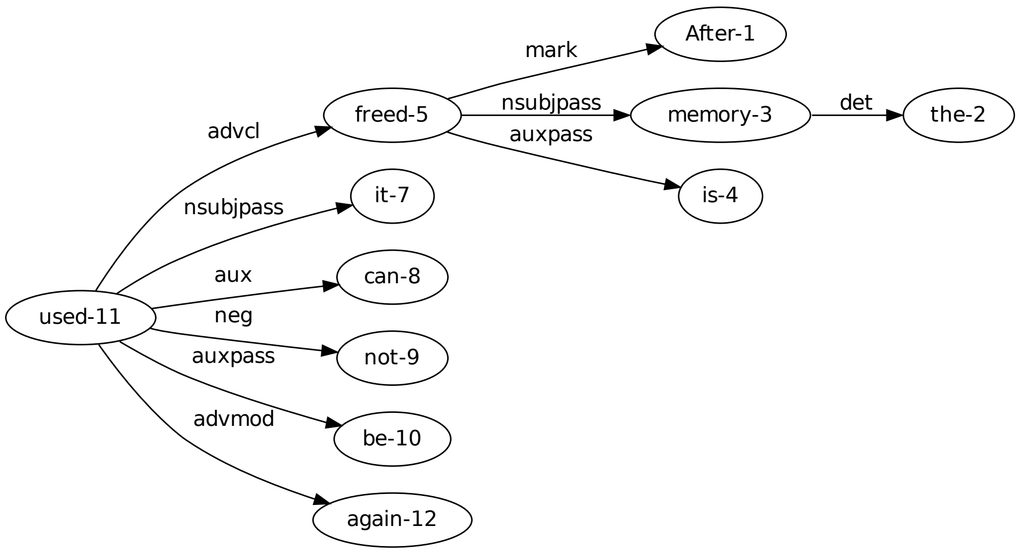
It clearly didn't understand the „again" in the source sentence, so the *G* temporal operator is not present in the LTL formula. Let's teach him:

1. We will use the tool `visualizer` (it can be found in the `tools` directory) that puts a sentence into The Standford Parser, gets the typed dependencies and visualizes it as a graph. We need to make sure that our system contains the `graphviz` package with the `dot` tool, and correctly configured JVM path in `config.py` at the root directory of the program.

2. Let's run the sentence through the visualizer:

   ```
   ./visualizer "Imediately after the memory is freed, it cannot \
   be used again."
   ```

   We'll get the following figure:

3. We can see that the word `again` is connected to the predicate of the second sentence by `advmod` typed dependency.

4. Now that we know the structure of the new feature of the sentence, we can write a rule. Append the following lines to the `pattern_directory.yaml`:

```
- cls: globally
  dependency_list:
  - dep: {type: value, value: again}
    reln: {type: value, value: advmod}
```

Let's break down the listing above, line by line:

- `cls: globally` says, that the translator should assign the tag `globally` to the analyzed sentence, if the sentence contains all of the further defined dependencies.
- `dependency_list:` just introduces an enumeration of the required dependencies.
- `dep: {type: value, value: again}` says, that the dependency's tail (*dep*endant) must be an exact value *"again"*.
- `reln: {type: value, value: advmod}` says, that the dependency's name must be an exact value *"advmod"*.

5. If we run the translator now, the sentence is translated correctly already:

```
(F(freed(memory)) -> XG(!used(memory)))
```

C.4.2 Adding a Synonym

Suppose we want the parser to learn to translate the following sentence:

```
Imediately after the memory is freed, it cannot ever be used.
```

It again didn't recognize the word „ever" implying global validity:

```
(F(freed(memory)) -> X(!used(memory)))
```

Proceeding as described in the previous section, after creating a visualization of the sentence through the `visualizer` tool, we realize, that the word „ever" is connected to the predicate by the same typed dependency. So let's create a new group of synonymes, and modify the rule from the previous section to work with it:

1. Append/modify the following in `pattern_directory.yaml`, which says to the pattern-matcher that instead of matching against a single value, to do a look-up in the synonym group dictionary:

```
- cls: globally
  dependency_list:
  - dep: {type: group, name: globally_adverbs}
    reln: {type: value, value: advmod}
```

Let's break down the listing above, line by line:

- `cls: globally` says, that the translator should assign the tag `globally` to the analyzed sentence, if the sentence contains all of the further defined dependencies.
- `dependency_list:` just introduces an enumeration of the required dependencies.

- `dep: {type: group, name: globally_adverbs}` says, that the dependency's tail (*dep*endant) must be one of the words from a group called *"globally_adverbs"*.
- `reln: {type: value, value: advmod}` says, that the dependency's name must be an exact value *"advmod"*.

2. Now, create a new synonym group called *globally_adverbs* in `synonym_groups.yaml`, and fill it with the „ever", „again" words:

```
globally_adverbs: [ever, again]
```

3. When we run the translator now, it translates the sentence correctly:

```
(F(freed(memory)) -> FG(!used(memory)))
```