TEMPORAL CONTROLLED NATURAL LANGUAGE FOR FORMAL SPECIFICATION

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

2017

By
Reyadh Alluhaibi
School of Computer Science

Contents

Word Count: 42832

List of Tables

List of Figures

Abstract

Temporal
Controlled Natural Language
for
Formal Specification
Reyadh Alluhaibi
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2017

Formal specifications describe behaviours and properties of embedded systems within a wide range of settings, i.e. from portable mobile devices, such as smartphones and tablets, to large stationary installations such as MRI machines and automobiles. These formal specifications are statements expressed in a temporal logic. Moreover, the formal specifications help to detect problems in system requirements by verifying the correctness of systems with respect to their formal specifications through automated reasoning tools. With the growing complexity of embedded system designs, industries employ formal specifications to verify the implementation of systems to reduce space, power and costs.

In recent years, *controlled natural language*, which is a subset of natural language with restricted syntax and semantics, is used to extract formal specifications from natural language specifications for embedded systems. *Controlled natural language* can reduce ambiguity and eliminate the complexity of natural language specifications. However, after the critical analysis of existing proposals in literature with regard to generating formal specifications from natural language specifications, we have observed that there is a lack of scalable tools for capturing temporal constructions in natural language specifications.

This thesis presents a scalable *controlled natural language* for generating formal specifications from natural language specifications featuring temporal constructions. These constructions are one of the greatest challenges in this field, since there are many obstacles

within the field of the temporal semantics of natural languages, such as tenses, aspects and temporal prepositions. We examine a variety of formal semantics of temporal constructions in literature. However, there are still many varieties of temporal expressions in English which these existing theories cannot account for.

Therefore, we exploit and extend an *interval temporal logic*, called *TPL*, to capture formally a range of temporal constructions that are often featured in natural language specifications. The logic *TPL* has variables which range over time-intervals and predicates corresponding to event-types and temporal order-relations. We then construct a transformation method that maps *TPL* into our desired formal specifications. *TPL* makes the transformations from natural language descriptions to formal specification possible since *TPL* has enough expressive power to deal with most temporal constructions commonly encountered in natural language specifications. We are able to prove the translation from *TPL* to formal specification theoretically. We also demonstrate the efficiency of our method in practice, which shows enormous performance improvements in comparison to known tools.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Dr. Ian Pratt-Hartmann for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me at all times while researching and writing this thesis. I could not have imagined having a better supervisor and mentor for my Ph.D study.

I would also like to extend my gratitude to my examiners Professor Allan Ramsay and Dr. Antony Galton for their valuable suggestions on how to improve my thesis, and those who donated their precious time to discuss various parts of my thesis with me, especially Adel Binbusayyis, Ayoade Adeniyi, Fabio Papacchin, Muhannad Almohaimeed, Patrick Koopmann, Thamer Ba-Dhafari and Yizheng Zhao.

A special thanks to my family. Words cannot express how grateful I am to my mother, and father for all of the sacrifices that you've made on my behalf. Your prayer for me was what sustained me thus far. They have been my inspiration and motivation for continuing to improve my knowledge and for moving my career forward.

At the end I would like express appreciation to my beloved wife Nuran who inspired me and provided constant encouragement during the entire process. Finally, I would also like to thank my three beautiful kids Salmaan, Sulaf and Safana who missed out on a lot of Daddy time while I sought intellectual enlightenment. I thank all four of you for your patience and love you more than you will ever know.

List of Abbreviations

The following is a list of various abbreviations used throughout the thesis.

ACE	Attempto Controlled English
AMBA	The Advanced Microcontroller Bus Architecture
CFG	Context Free Grammar
CNL	Controlled Natural Language
CTL	Computation Tree Logic
DFA	Deterministic Finite-state Automaton
DRS	Discourse Representation Structures
D-TPE	Direct TempCNL Parsing Engine
DSVA_AMBA	SVA taken from AMBA dataset for the D-TPE system
DSVA_OCP	SVA taken from OCP dataset for the D-TPE system
FOL	First-Order Logic
ITL	Interval Temporal Logic
LTL	Linear Temporal Logic
LTL^+	LTL with only the future operators
OCP	Open Core Protocol
OWL	Web Ontology Language
SVA	SystemVerilog assertions
SVA_AMBA	SVA taken from AMBA dataset for the TPE system
SVA_OCP	SVA taken from OCP dataset for the TPE system
TPL	Temporal Preposition Logic
TPL^+	An extension of Temporal Preposition Logic
TempCNL	Temporal Controlled Natural Language
TPE	TempCNL Parsing Engine

Chapter 1

Introduction

Formal specifications are used to describe the requirements of computer hardware in diverse environments. They build on end-user requirements which demand consistent performance, reliability, and integration of systems. Formal specifications are like processes in some cases. If a company is planning to build a computer network, formal specifications would be a key step in that company's planning.

In recent years, there has been an increasing interest in formal verification techniques (Darringer, 1988; Milne, 1993) for checking whether a design fulfils certain requirements (properties). Verification techniques —such as theorem proving or model checking— are used to prove or disprove the correctness of a system design with respect to formal specifications expressed in the appropriate formal language, such as *computation tree logic* (henceforth *CTL*), *linear temporal logic* (henceforth *LTL*) and *SystemVerilog assertions* (henceforth *SVA*). These languages are, in effect, forms of temporal logic, which have formed a significant research subject in hardware designs. They have a proven track-record in industrial applications.

Most such specifications must be written manually which leads to various practical difficulties such as (i) the time spent in generating formal specifications, (ii) the number of design errors, (iii) incomplete and inconsistent specifications, and (iv) the cost of training users for writing formal specifications which require high levels of mathematical and analytical skills to formulate the complexities of hardware designs such as system-on-a-chip (SoC), the properties of the design of which is difficult to verify in clear and precise ways.

We therefore employ a so-called *controlled natural language* (henceforth *CNL*) to specify system behaviour. *CNL* will help to eliminate all the previously mentioned limitations by decreasing the time spent in generating formal specifications, reducing the number of design errors, and identifying incomplete and inconsistent specifications in the early stages of planning. *CNL* is a subset of natural language with a restricted syntax and semantics defined by a set of syntax and semantic rules. *CNL* texts are processable by computer and can be translated into formal representations such as *discourse representation structures* (henceforth *DRS*) and *first-order logic* (henceforth *FOL*), and *Web Ontology Language* (henceforth *OWL*). For example, *Attempto Controlled English* (henceforth *ACE*) is a *CNL* introduced in Fuchs and Schwitter (1996). *ACE* has a tool called *Attempto Parsing Engine* which helps users to represent their texts into *DRS*, *FOL*, and *OWL*.

CNL has been used in the areas of software and hardware specifications (e.g. Fuchs and Schwitter, 1996; Grover et al., 2000), specifications of legal contracts (e.g. Pace and Rosner, 2009; Wyner et al., 2016), text summaries (e.g. Kuhn et al., 2006), business rule specifications (e.g. Ross, 2003; Bajwa et al., 2011), and interfaces to formal ontologies (e.g. Funk et al., 2007). Any *CNL* must include the following vocabulary types: predefined words (such as determiners, prepositions, conjunctions etc.) and content words (such as nouns, verbs, adjectives, and adverbs etc.). Moreover, *CNL* is constructed from a grammar which consists of a small set of syntactic and semantic rules. The syntax rules determine the acceptable sentence structures and also avoid ambiguous or imprecise syntax. The semantic rules define the meanings of *CNL* texts, which requires first controlling logical analysis of acceptable sentence and second, solving remaining ambiguities.

Various attempts have been made to generate formal system requirements from natural language documents such as (Clarke et al., 1986; Osborne and MacNish, 1996; Holt, 1999;

Grover et al., 2000; Lamar, 2009; Harris, 2013). However, generating formal requirements from natural language specifications is still limited because (1) most such approaches fail to cope with natural language specifications featuring temporal constructions such as tenses, aspects, and temporal prepositions, and (2) hardware languages such as *SVA* and *LTL* are less expressive than natural languages, thus, these approaches are inadequate for handling the complexity of natural languages when they are mapped directly into hardware languages.

Accordingly, we review several studies on providing the formal semantics for temporal expressions in natural language (e.g. Reichenbach, 1947; Prior, 1967; Davidson, 1967; Vendler, 1967; Dowty, 1972; Steedman, 1977; Moens, 1987; Parsons, 1990; Ogihara, 1994; Pratt and Francez, 2001). In the history of the development of the temporal representations of natural language, tense, aspect and temporal preposition phenomena have become central issues for this area. Tense logic (Prior, 1967) was one of the first attempts to capture temporal expressions in natural language. Prior represents the meaning of sentences as having tense based on specific points of the timeline. The definition of the timeline is a linearly ordered set of points that is unbounded at both ends. Lyons (1977) describes tense as a deictic category which will contain a reference to some point or period of time that is only identifiable in the zero-point of the utterance.

The second area of temporal semantics we consider is aspect. During the last decade, linguists such as Vendler (1967), Dowty (1972), Steedman (1977) and Moens (1987) have worked to build a taxonomy of temporal-event descriptions to provide better descriptions of how people describe events in our language. Vendler (1967) began the study of the internal structure of temporal-events and the way in which language users can describe various subparts of events based on the sentences or discourses in which they appear. Understanding aspectual class is essential for giving us the proper interpretations for the internal structure of temporal-events.

The final topic in temporal semantics of natural language that we consider is temporal prepositions (such as *once*, *until*, *when* etc.). The temporal prepositions commonly occur in natural language specifications. In recent years, Ogihara (1994) and Pratt and Francez (2001) have provided formal representations for temporal prepositions in English. Temporal prepositions express the relations between events which add a level of complexity to the semantics of sentence. In another study, Pratt-Hartmann (2005) introduced an *interval temporal logic* called TPL. This logic can capture formally most temporal constructions that are described in natural language specifications. TPL is a first-order language having variables which range over time-intervals, and predicates corresponding to event-types and temporal order-relations.

1.1 Research goals

As mentioned above, most of the current approaches, such as (Clarke et al., 1986; Holt, 1999; Grover et al., 2000; Harris, 2013), fail to capture certain temporal constructions commonly occurring in natural language specifications such as aspects and temporal prepositions. Therefore, our goal in this study is to implement a *CNL* to generate formal specifications from natural language specifications. Using a *CNL* helps reduce ambiguity and eliminate the complexity of natural language specifications since it is constructed with a restricted syntax and semantics.

In this study, we choose to dedicate our *CNL* for generating SVA which is one of the preferred languages for writing formal specifications. SVA is a subset of SystemVerilog, which combines hardware descriptions and formal verifications. SVA can validate the behaviour of a design dynamically using a simulator (such as Cadence in Simulator (2005), Modelsim in Graphics (2015), and VCS in Synopsys (2004)). SVA is a *linear temporal logic* which can express complex temporal behaviours of the system designs in a concise and accurate way.

In order to capture the semantics of temporal constructions in natural language specifications, we will use TPL, introduced in Pratt-Hartmann (2005), since theories of natural language temporal semantics are most naturally interval-based. Moreover, TPL is expressive enough to deal formally with the semantics of temporal constructions – such as prepositions – in natural language specifications. However, TPL must be extended to capture more temporal constructions in English. Using TPL with some extensions makes the transformations from natural language specifications to *SVA* possible.

But translating TPL to *SVA* in turn leads to another difficulty, since TPL is an interval-based logic, whereas *SVA* (which is essentially *LTL*) is a point-based logic. Such translations cause a major theoretical challenge because (1) TPL and *SVA* have different syntactic and semantic notions, and (2) interval-based logics are already known to be more expressive than point-based ones. We therefore need to find a convenient way to express TPL into *SVA*.

The main contributions of this study are as follows:

- To implement a CNL that can capture temporal expressions in English.
- To extend TPL to capture more temporal constructions in English.
- To define a context-free grammar for extracting our extended TPL formulas.
- To construct transformation rules for generating *SVA* from our extended TPL.
- To improve the accuracy rate of generating *SVA* from natural language specifications, compared to existing tools.

1.2 Thesis outline

The structure of this thesis is as follows. Chapter 2 gives technical background on Montague semantics, *linear temporal logic*, *SystemVerilog assertions* and *interval temporal logic*. In Chapter 3, we present an overview of formal temporal semantics in natural languages by explaining what methods have been developed in the literature to capture temporal semantics from texts. In Chapter 4, we present a prototype *CNL* that extracts temporal expressions from raw text. Chapter 5 provides transformation methods for generating *linear temporal logic* and *SystemVerilog assertions* from our extended TPL. Chapter 6 describes the evaluation of our *temporal CNL* for generating *SystemVerilog assertions* from natural languages specifications. Finally, in Chapter 7, we conclude and discuss future work.

Chapter 2

Technical Background

In this chapter we present the technical background that is used in the rest of the thesis. Section 2.1 imparts the basic concepts of Montague semantics for building semantic representations for a fragment of English. Section 2.2 describes the top-down parsing algorithm which will be used to parse our CNL sentences. Section 2.3 presents the syntax and the semantics of *linear temporal logic*. Section 2.4 shows what sort of sentences we are dealing with in natural language specifications. Section 2.5 presents the syntax and the semantics of *SystemVerilog assertions*. In final section, we give the syntax and the semantics of *interval-temporal logic*.

2.1 Montague Semantics

Montague (1974) connected the syntactic structure and semantic structure of natural language in a way that allows for a better understanding of the semantic meanings of sentences. Nowadays, many approaches use Montague semantics to build their semantic representations for a fragment of English. In this section, we will show how we automatically translate sentences from English into *first-order logic* (henceforth *FOL*) based on Montague's method which uses *context free grammars* (henceforth *CFG*) that combines typed logic with lambda abstraction.

Before we start to look at the Montague semantics in more detail, we begin by introducing *CFG*. The following definition is proposed by Hopcroft and Ullman (1979).

Definition 2.1.1. A *context-free grammar* G is a tuple (N, Σ, P, S), where N is a finite set of *non-terminal symbols* such as S, NP, and VP; Σ is a finite set of *terminal symbols*; P is a set of (*phrase structure*) *rules* of the form $\alpha \rightarrow \beta$, where $\alpha \in N$ and β is a string of symbols from $N \cup \Sigma$, and $S \in N$ is the *start symbol.*

We can generate strings of the S category by recursively evaluating the rules of G as follows. If $\alpha \in N$ is a non-terminal symbol, $S_1, ..., S_m \in N \cup \Sigma$ and $\alpha \rightarrow S_1, ..., S_m \in P$ is a rule in G, then α evaluates to the sequence of the evaluations of $S_1, ..., S_m$. If some $S_i \in N$, $1 \leq i \leq m$, then evaluation of S_i proceeds recursively. Every terminal symbol evaluates to itself. A string of category S is then a result of evaluating S in G. For example, Let $G_1 = (\{S, T, O, I\}, \{0, 1\}, P, S)$, where P contains the following productions:

$$\begin{aligned} \mathrm{S} &\rightarrow \mathrm{OT} \\ \mathrm{S} &\rightarrow \mathrm{OI} \\ \mathrm{T} &\rightarrow \mathrm{SI} \\ \mathrm{O} &\rightarrow 0 \\ \mathrm{I} &\rightarrow 1 \end{aligned}$$

The above grammar can be used to describe the set $\{0^n 1^n \mid n \geq 1\}$, and the string "000111" is a sentence generated by G_1.

For the English language, we can simply represent a grammar G = (N, Σ, P, S) via a given a list of rules P, as shown in Figure 2.1.1, where the start symbol, in the given example, is S. Moreover, Figure 2.1.1 shows the phrase structures of some sentences that are generated by this grammar.

$$\begin{array}{rcl}
\text{S} & \rightarrow & \text{NP, VP} \\
\text{NP} & \rightarrow & \text{Det, N} \\
\text{NP} & \rightarrow & \text{PN} \\
\text{VP} & \rightarrow & \text{V, NP} \\
\text{Det} & \rightarrow & \text{every}|\text{some}|\text{a} \\
\text{N} & \rightarrow & \text{boy}|\text{girl} \\
\text{PN} & \rightarrow & \text{John}|\text{Mary} \\
\text{V} & \rightarrow & \text{loves}
\end{array}$$

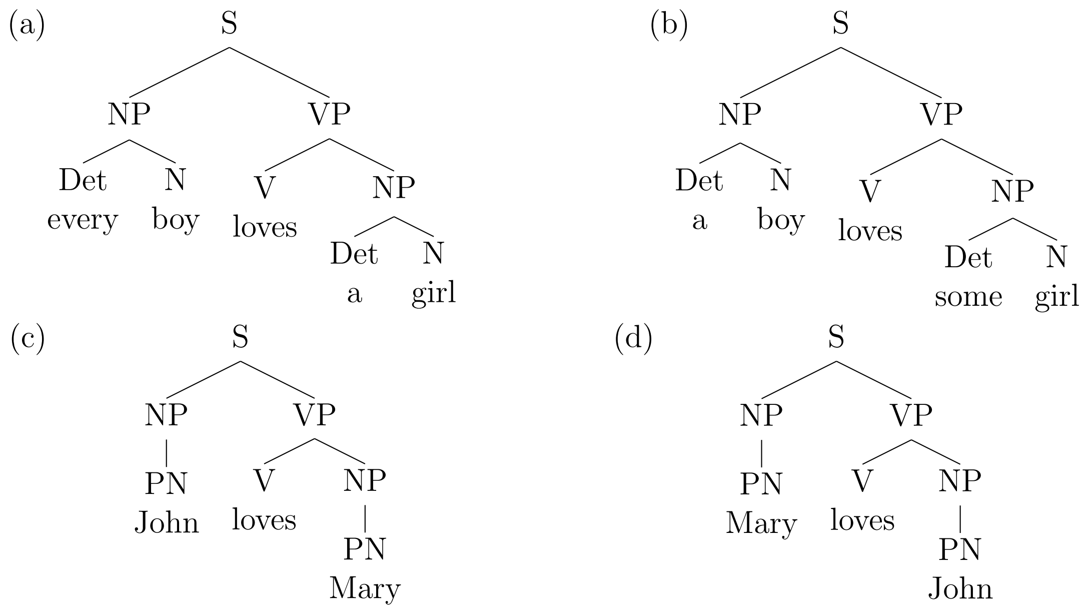


Figure 2.1.1: A sample grammar and the structures of some sentences that are generated by it.

In the grammar in Figure 2.1.1, the category coressponding to the star symbol is sentence (*S*), which consists of a noun phrase (*NP*) followed by a verb phrase (*VP*). The *NP* consists of either a proper noun (*PN*) or a determiner (*Det*) followed by a noun (*N*). Returning to the verb phrase (*VP*), it consists of a verb (*V*) followed by an *NP*.

Every English grammar is divided into three parts. The first part is called the *syntax* which is the set of rules with non-terminal symbols on the right-hand side. The second and third parts are called the formal lexicon and the content lexicon which both describe the set of rules with only terminal categories on the right-hand side. The formal lexicon has rules extracting the closed class of grammaticalised words such as *every*, *some*, *a* and *not*. However, the content lexicon also has rules for extracting the open classes of nouns and verbs.

Now, let us turn to how we can extract the semantics of sentences using Montague semantics. Montague (1974) provides an effective way to extract the meaning of fragments of English into intensional logic. He makes this possible by using a context-free grammar that combines typed logic with lambda abstraction. Montague (1974) defines the semantics of λ-expressions for generating the semantics of sentences as follows.

Definition 2.1.2. Suppose we have a set of *types*, consisting of basic type e (the type of entities) and t (the type of truth values) and *functional types*, where a functional type is of the form $\langle \tau_1, \tau_2 \rangle$ for some basic or functional types τ_1, τ_2.

Let μ be a first-order signature, and let a *typed variable* be a symbol not occurring in μ which has an associated type τ.

The set of λ-expressions over a given signature μ and set T of typed variables is constructed recursively by the consecutive conditions:

1. Every first-order term over the signature μ and every element of T of type e is a λ-expression of type e.

2. Every closed first-order formula over the signature μ and every element of T of type t is a λ-expression of type t.

3. If ϕ is a λ-expression of type τ_2, and x is a variable of type τ_1 in T possibly taking place in ϕ, then $\lambda x[\phi]$ is a λ-expression of type $\langle \tau_1, \tau_2 \rangle$. In this case we say that λx binds x in ϕ.

4. If ψ is a λ-expression of type τ_1, and ϕ is a λ-expression of type $\langle \tau_1, \tau_2 \rangle$, then $\phi(\psi)$ is a λ-expression of type τ_2. In this condition we say that ϕ is applied to ψ.

Let us now give an example of how we compute a *FOL* formula with λ-expressions as follows: let q have type $\langle e,t\rangle$, let x have type e and let *boy*, *human* be unary predicates. Then the λ-expressions $\lambda y[human(y)]$ and $\lambda q[\forall x(boy(x) \rightarrow q(x))]$ has types $\langle e,t\rangle$ and $\langle\langle e,t\rangle,t\rangle$ respectively, as can be seen if we apply the latter to the former and compute the β-reduction. We begin with

$$\lambda q[\forall x(boy(x) \rightarrow q(x))](\lambda y[human(y)]),$$

and substitute every occurrence of q in $\forall x(boy(x) \rightarrow q(x))$ with $\lambda y[human(y)]$, to get

$$\forall x(boy(x) \rightarrow (\lambda y[human(y)])(x)),$$

This expression has another instance of λ-expression application, and thus we β-reduce again, to obtain

$$\forall x(boy(x) \rightarrow human(x)),$$

where it has type t, as required.

Observe that, to prevent any possible clashes arising from two λ-expressions containing common variables, we assume, in every instance of β-reduction, that neither formula includes any typed variables taking place in the other. We quietly substitute any expressions violating this case with an α-equivalent expression wherever needed.

Now, we get to Montague semantics for extracting the semantics of words in English. We can use λ-expressions to extract the semantics of a phrase from the semantics of its components. For instance, if *John* has the semantics $\lambda p[p(john)]$, and *walks* has the semantics $\lambda x[walks(x)]$, then the semantics of *John walks* can be computed by applying the semantics of *John* to the semantics of *walks*, to produce *walks(john).*

We represent the assignment of semantics to words and phrases through *semantically annotated grammars.* Suppose a grammar G includes a rule of the form $R \rightarrow R_1$, R_2, and a semantically annotated version G' of G includes a rule of the form $R/\psi(\phi) \rightarrow R_1/\phi$,

R_2/ψ. Then G' is interpreted as stipulating that if R_1, R_2 are assigned semantics ϕ, ψ, then the semantics of R are computed by applying the semantics of R_2 to the semantics of R_1. The real values of ϕ, ψ can be computed by recursively evaluating R_1, R_2 by additional annotated rules of G'. If R_1 is a terminal category, then the meaning of ϕ is a λ-expression.

Figure 2.1.2 shows how the grammar of Figure 2.1.1 can be annotated with compositional semantics in this manner.

$$
\begin{array}{l}
\mathrm{S}/\phi(\psi) \rightarrow \mathrm{NP}/\phi,\ \mathrm{VP}/\psi \\
\mathrm{NP}/\phi(\psi) \rightarrow \mathrm{Det}/\phi,\mathrm{N}/\psi \\
\mathrm{NP}/\phi \rightarrow \mathrm{PN}/\phi \\
\mathrm{VP}/\phi(\psi) \rightarrow \mathrm{V}/\phi,\mathrm{NP}/\psi \\
\mathrm{Det}/\lambda q\lambda p[\exists x(q(x) \wedge p(x))] \rightarrow \text{some} \mid \text{a} \\
\mathrm{Det}/\lambda q\lambda p[\forall x(q(x) \rightarrow p(x))] \rightarrow \text{every} \\
\mathrm{N}/\lambda x[boy(x)] \rightarrow \text{boy} \\
\mathrm{N}/\lambda x[girl(x)] \rightarrow \text{girl} \\
\mathrm{PN}/\lambda p[p(john)] \rightarrow \text{John} \\
\mathrm{PN}/\lambda p[p(mary)] \rightarrow \text{Mary} \\
\mathrm{V}/\lambda s\lambda x[s(\lambda y[love(x,y)])] \rightarrow \text{loves}
\end{array}
$$

Figure 2.1.2: A sample annotated grammar.

The rule $\mathrm{S}/\phi(\psi) \rightarrow \mathrm{NP}/\phi,\ \mathrm{VP}/\psi$ in Figure 2.1.2 is interpreted as follows: a sentence consists of a noun phrase and a verb phrase. If the meaning of the noun phrase is ϕ and the meaning of the verb phrase is ψ, then the meaning of the sentence is ψ applied to ϕ. Other rules are interpreted similarly. Figure 2.1.3 shows how the annotated grammar of Figure 2.1.2 maps a range of English sentences into formulas of first-order logic.

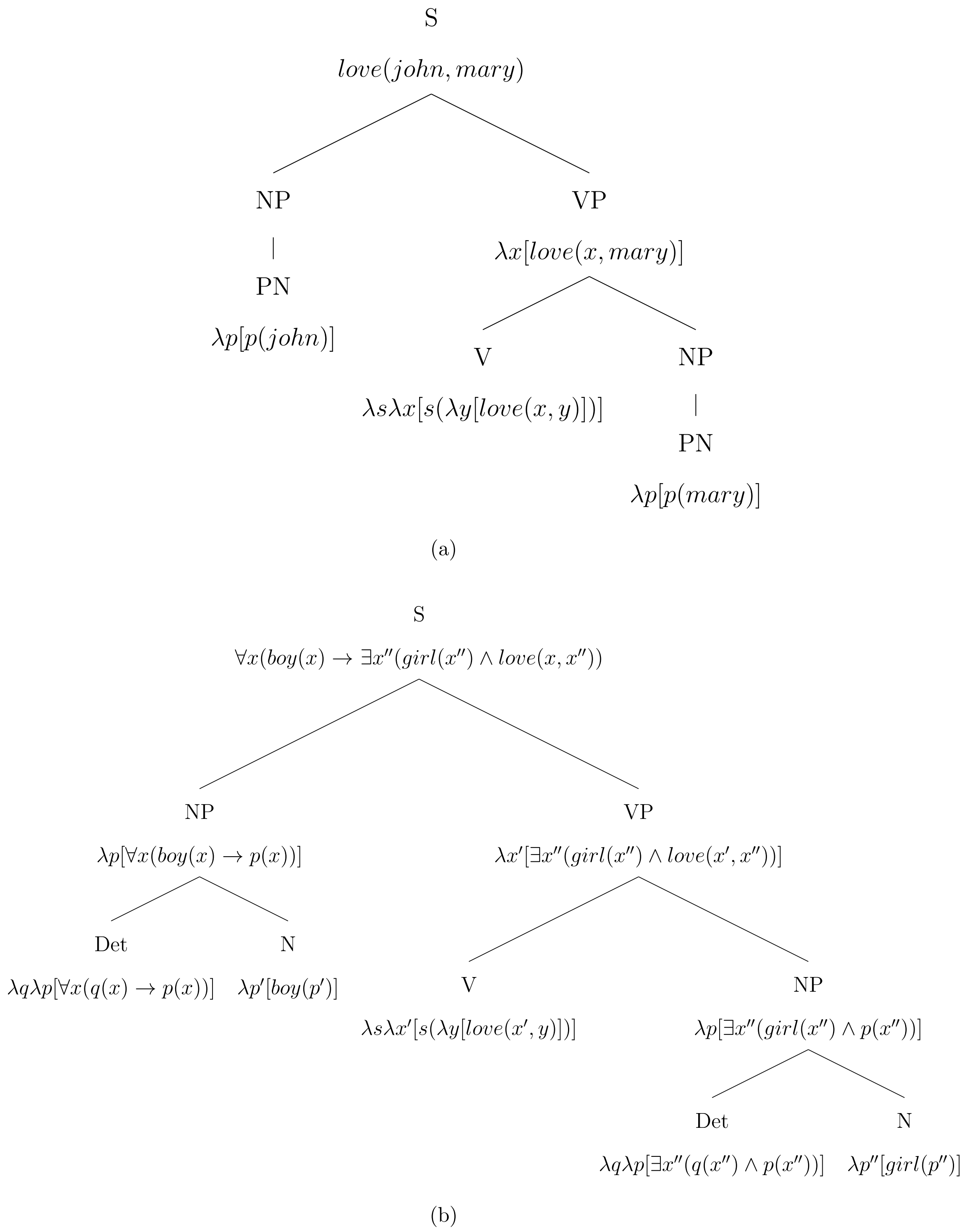

S
$love(john, mary)$
NP
PN
$\lambda p[p(john)]$
VP
$\lambda x[love(x, mary)]$
V
$\lambda s \lambda x[s(\lambda y[love(x, y)])]$
NP
PN
$\lambda p[p(mary)]$
(a)
S
$\forall x(boy(x) \rightarrow \exists x''(girl(x'') \wedge love(x, x''))$
NP
$\lambda p[\forall x(boy(x) \rightarrow p(x))]$
Det
$\lambda q \lambda p[\forall x(q(x) \rightarrow p(x))]$
N
$\lambda p'[boy(p')]$
VP
$\lambda x'[\exists x''(girl(x'') \wedge love(x', x''))]$
V
$\lambda s \lambda x'[s(\lambda y[love(x', y)])]$
NP
$\lambda p[\exists x''(girl(x'') \wedge p(x''))]$
Det
$\lambda q \lambda p[\exists x''(q(x'') \wedge p(x''))]$
N
$\lambda p''[girl(p'')]$
(b)

Figure 2.1.3: Computing semantics from an annotated grammar in Figure 2.1.2.

We have shown Montague semantics that preform operations on functions and entities. However, an alternative possibility is to use Montague semantics to preform operations on strings. The idea is that the λ-expressions denote functions mapping strings to strings. For example, we might have

$$[\![\text{runs}]\!] = \lambda x[\text{“run(”} + x + \text{“)”}].$$

Note that + denotes string concatenation here. Moreover, here the string x is mapped to the concatenation of the strings “run(”, x itself and “)”. Thus,

$$\begin{aligned}[\![\text{runs}]\!](\text{“john”}) = {} & \text{“run(”} + \text{“john”} + \text{“)”}. \\ = {} & \text{“run(john)”}.\end{aligned}$$

If, then

$$[\![\text{John}]\!] = \lambda p[\text{p(“john”)}],$$

we have

$$\begin{aligned}[\![\text{John}]\!]([\![\text{runs}]\!]) = {} & \lambda p[\text{p(“john”)}]\underline{(\lambda x[\text{“run(”} + x + \text{“)”}])}. \\ = {} & \lambda x[\text{“run(”} + x + \text{“)”}]\underline{(\text{“john”})}. \\ = {} & \text{“run(”} + \text{“john”} + \text{“)”}. \\ = {} & \text{“run(john)”}.\end{aligned}$$

Similarly, if

$$\begin{aligned}&[\![\text{every}]\!] = \lambda p\lambda q[\text{“}\forall x\text{(”}+p(\text{“}x\text{”})+\text{“}\rightarrow\text{”}+q\ (\text{“}x\text{”})+\text{“)”}], \\ &[\![\text{boy}]\!] = \lambda x'[\text{“boy(”}+x'+\text{“)”}] \text{ and} \\ &[\![\text{loves some girl}]\!] = \lambda x'[\text{“}\exists y\ (\text{girl}(y) \wedge \text{love(”}+x'+\text{“,}y\text{))”}].\end{aligned}$$

then we first compute the meaning of “every boy” as follows:

$$\begin{aligned}[\![\text{every}]\!]([\![\text{boy}]\!]) = {} & \lambda p\lambda q[\text{“}\forall x\text{(”}+p(\text{“}x\text{”})+\text{“}\rightarrow\text{”}+q(\text{“}x\text{”})+\text{“)”}]\underline{(\lambda x'[\text{“boy(”}+ x' +\text{“)”}])}. \\ = {} & \lambda q[\text{“}\forall x\text{(”}+\lambda x'[\text{“boy(”}+ x'+\text{“)”}]\underline{(\text{“}x\text{”})}+\text{“}\rightarrow\text{”}+q\ (\text{“}x\text{”})+\text{“)”}]. \\ = {} & \lambda q[\text{“}\forall x\text{(”} +\text{“boy(”}+\text{“}x\text{”}+\text{“)”}+\text{“}\rightarrow\text{”}+q(\text{“}x\text{”})+\text{“)”}]. \\ = {} & \lambda q[\text{“}\forall x(\text{boy}(x) \rightarrow\text{”}+q\ (\text{“}x\text{”})+\text{“)”}].\end{aligned}$$

Secondly, we compute the meaning of “every boy loves some girl” by combining the meaning of “every boy” with the meaning of “loves some girl” as follows:

$$\begin{aligned}
[\![\text{every boy}]\!]([\![\text{loves some girl}]\!]) &= \lambda q[\text{“}\forall x(\text{boy}(x) \rightarrow\text{”}+q(\text{“}x\text{”})+\text{“})\text{”}]\underline{(\lambda x'[\text{“}\exists y(\text{girl}(y)\wedge \text{ love(”}+x'+\text{“},y))\text{”}])}\\
&= \text{“}\forall x(\text{boy}(x) \rightarrow\text{”}+(\lambda x'[\text{“}\exists y(\text{girl}(y)\wedge \text{ love(”}+x'+\text{“},y))\text{”}])\ \underline{(\text{“}x\text{”})}\ +\text{“})\text{”}\\
&= \text{“}\forall x(\text{boy}(x) \rightarrow\text{”}+\text{“}\exists y(\text{girl}(y)\ \wedge\ \text{love(”}+\text{“}x\text{”}+\text{“},y))\text{”}+\text{“})\text{”}\\
&= \text{“}\forall x(\text{boy}(x) \rightarrow \exists y(\text{girl}(y)\wedge \text{ love}(x,y)))\text{”}.
\end{aligned}$$

The style of semantics is conceptually less satisfying Montague’s; however, it sometimes saves effort, as we shall see in Chapter 4.

Making a concession to informality we omit the “+”s and quotes, but writing @ for function application. For example

$$[\![\text{every}]\!] = \lambda p\lambda q[\forall x(p@x \rightarrow q@x)],$$

and so on.

In summary, using Montague semantics with *CFG* enables us to extract the semantics of our extended TPL formulas in a similar way, except, of course, we need to annotate grammars with the corresponding semantics of that logic. In this section, we have shown how to extract the semantics of a small fragment of English. Moreover, we have shown that Montague semantics can preform operations not only on functions and objects but also on strings. It should be borne in mind that we have presented only simple syntax which do not include tense and aspect.

2.2 Top-Down Parsing Algorithm

Parsing (syntactic analysis) is a process that constructs automatically a syntactic structure (i.e. parse tree) from an input sentence in terms of a given grammar and lexicon in order to determine its grammar structure. Occasionally, the resulting syntactic analyses are used as input to a process of semantic interpretation. There are various parsing algorithms that developed in last century such as top-down parsing, bottom-up parsing, left-corner parsing, chart parsing, and shift-reduce parsing. These algorithms can be run either using depth-first or breadth-first search strategy.

In this thesis, we use the top-down parsing algorithm to construct the parse trees from the input sentences in our controlled natural language as we shall see in Chapter 4. This section describes this algorithm and how it can obtain the parse trees using depth-first search. We will use an example sentence to illustrate and describe the way it process sentences to produce parse trees. Finally, this section shows some limitations of the top-down parsing algorithm in term of its performance, and how we can possibly avoid some of these limitations.

The top-down parsing algorithm splits up a sentence into words and phrases until the phrase structure for a given sentence is generated. The process of this algorithm starts from top of a parse tree with the start symbol (S) and expands it downwards by employing and expanding rules of a given grammar until it reaches the terminal symbols (Aho and Ullman, 1972, p. 285). Top-down algorithms can produce the correct syntactic parse tree for a sentence based on a set of grammar rules of a natural language.

We use the grammar rules in Figure 2.1.1 to parse the sentence “John loves Mary” as shown in Figures 2.2.1 to illustrate the parsing process in the top-down algorithm using depth-first search. Note that the parsing steps in each figure are presented using numbers to indicate the parser’s steps.

In Figure 2.2.1, a top-down parser uses depth-first search to explore recursively the left branch of the tree until it finds the target (which is an input word or phrase). Then, it would explore the right branch of the tree. However, if any of the selected branches fails, then the parser needs to backtrack and explores the alternative(s), and follows the same strategy. A depth-first search explores one branch at a time. As shown in Figure 2.2.1, after the root (S) of the parse tree is constructed, the search goes down to the left branch first, which toward the NP category since it is the first element of the S category in the

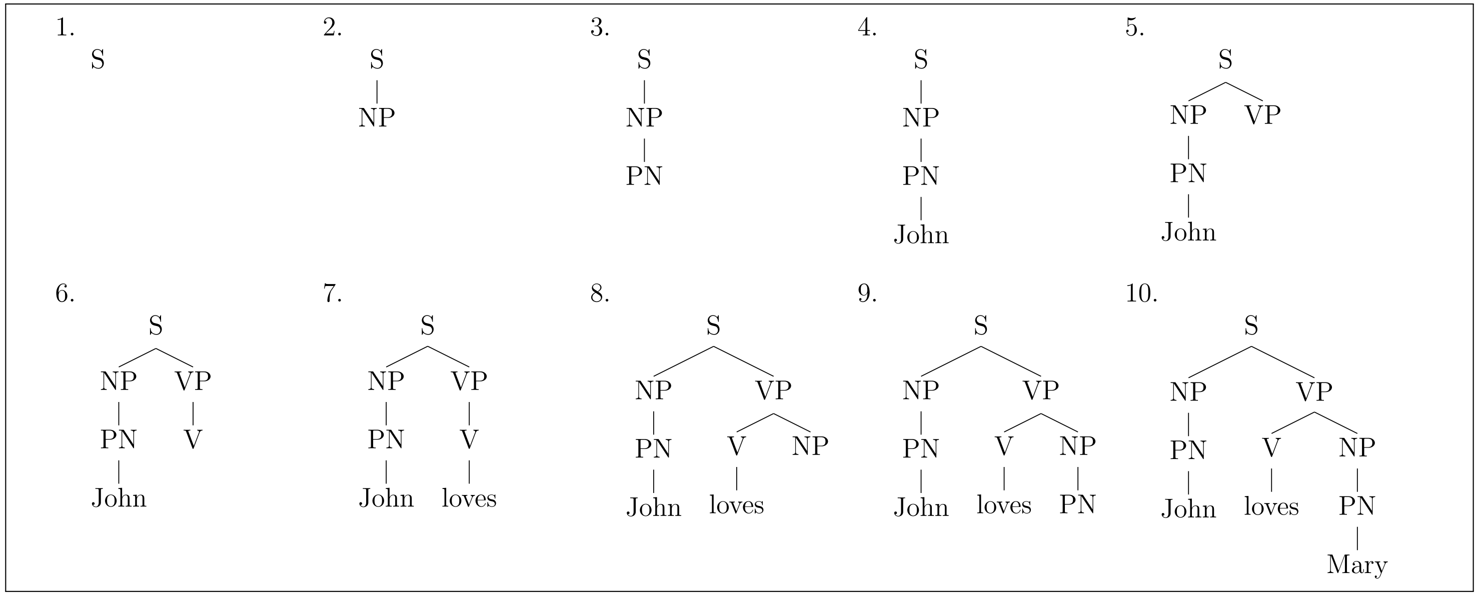


Figure 2.2.1: Top-down depth-first parsing.

grammar rules in Figure 2.1.1. Therefore, the parser creates a new partial parse tree as shown in 2.2.1(2). Next, the search continues going down further deep into the lower level by processing the PN category based on the grammar rules in 2.1.1. The PN category is a pre-terminal category which consists the word "John" on its right-hand side. At this stage, the search reaches the first word of the given sentence. Now, we move to the right branch of the root (S) of the parse tree and follow the same steps of the left branch until the parser generates the complete parse tree for the given sentence as shown in 2.2.1(10).

In practice, we prefer depth-first strategy over breadth-first strategy because the depth-first strategy requires an amount of memory that is proportional to the size of the problem (with respect to the given input sentences) while breadth-first search may require exponential memory since it keep tracks of all the nodes on the path from the root to the current node. Moreover, Prolog language, which we require to build our tool, uses a top-down depth-first parsing algorithm in order to define a large number of grammars in very elegantly. Thus, the code of a top-down depth-first parser can be smaller than other types of parsers. Therefore, we choose to build our controlled natural language using the top-down depth-first parser.

Now let us discuss some limitations of the top-down parsing algorithm using depth-first strategy as follows:

- This parsing algorithm is basically rule-driven because it does not have any knowledge of the the terminal symbols (words). Therefore, when it discovers that the current terminal symbol cannot be used as a production for the chosen pre-terminal symbol, it backtracks to explore alternative analyses that can be used as a production for the pre-terminal symbols in question. However, backtracking may impact the parser's performance since it requires the parser to repeat tasks that had already been done before backtracking since all the productions prior to the backtracking stage are lost (Chapman, 1987, p. 22). Specifically, a top-down parser requires an exponential number of steps (with respect to the length of the sentence) to try all the alternative analyses for producing a full parse tree for a complete sentence.

- It can not handle left recursion rules such as the following grammar rules:

$$\begin{array}{rcl} \text{NP} & \rightarrow & \text{NP, PP} \\ \text{NP} & \rightarrow & \text{Det, N} \\ \text{PP} & \rightarrow & \text{P, NP} \end{array}$$

Figure 2.2.2: Simple grammar rules with left-recursive.

where the first rule has the NP category in right and left hand sides, which is not allow in this type of parsing algorithm. The reason is that because it does not guarantee to find a solution even if there is one exists since it is possible to proceed down an infinitely long branch, without ever returning to explore other branches. However, there is alternative way to avoid left-recursive rules in this parsing algorithm and that is by transforming the grammar to a weakly equivalent non-left-recursive ones as shown in Figure 2.2.3, where it derives the same set of sentences as the grammar rules in Figure 2.2.2. This approach is well-known (see e.g. Moore, 2000).

$$\begin{array}{rcl} \text{NP}_1 & \rightarrow & \text{NP}_0\text{, PPS} \\ \text{NP}_1 & \rightarrow & \text{NP}_0 \\ \text{NP}_0 & \rightarrow & \text{Det, N} \\ \text{PPS} & \rightarrow & \text{PP, PPS} \\ \text{PPS} & \rightarrow & \varepsilon \\ \text{PP} & \rightarrow & \text{P, NP}_1 \end{array}$$

Figure 2.2.3: Simple grammar rules with non-left-recursive.

2.3 Linear Temporal Logic (LTL)

LTL was introduced by Pnueli (1977). A *LTL* formula express a property of a linear sequence of states. Any actual sequence of states may or may not match that property. For instance, the fact that properties such as p hold at some state in the sequence or p

holds at two consecutive states in the future can be expressed in *LTL*.

The *LTL* language consists of a finite set of atomic propositional variables P, the usual Boolean connectives, i.e. $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, and the future temporal operators G ('always'), F ('eventually'), U ('until'), and X ('next'), and as well as the past temporal operators H ('always in the past'), F^{-1} ('once in the past'), S ('since'), and X^{-1} ('next in the past'). We define *LTL*-formula ϕ in the following Backus-Naur form, where P represents an atomic formula.

$$\phi := P \mid \neg\phi \mid G\phi \mid F\phi \mid X\phi \mid \; U(\phi', \phi) \mid H\phi \mid F^{-1}\phi \mid X^{-1}\phi \mid \; S(\phi', \phi) \mid \phi \wedge \phi' \mid \phi \vee \phi'.$$

Before giving the formal semantics of LTL formulas, we will define the notion of LTL interpretation.

Definition 2.3.1. In LTL interpretation $\mathfrak{L}$ is a function mapping propositional letters to subsets of $\mathbb{N}$ (the natural numbers).

Let us now turn to the definition of the truth-conditions for formulas in LTL.

Definition 2.3.2. Let α be a formula and $t \in \pi$. We write $\mathfrak{L} \models_t \alpha$ to denote that "α is true at time instant t in the model $\mathfrak{L}$". This notion is defined recursively, according to the structure of α.

1. For $p \in P$, $\mathfrak{L} \models_t p$ iff $t \in \mathfrak{L}(p)$;

2. $\mathfrak{L} \models_t \neg\alpha$ iff $\mathfrak{L} \models_t \alpha$ is false;

3. $\mathfrak{L} \models_t \alpha \wedge \alpha'$ iff both $\mathfrak{L} \models_t \alpha$ and $\mathfrak{L} \models_t \alpha'$;

4. $\mathfrak{L} \models_t \alpha \vee \alpha'$ iff $\mathfrak{L} \models_t \alpha$ or $\mathfrak{L} \models_t \alpha'$;

5. $\mathfrak{L} \models_t G\alpha$ iff for all $t' \geq t$, $\mathfrak{L} \models_{t'} \alpha$

6. $\mathfrak{L} \models_t F\alpha$ iff for some $t' \geq t$, $\mathfrak{L} \models_{t'} \alpha$

7. $\mathfrak{L} \models_t U(\alpha', \alpha)$ iff some $t' \geq t$ such that $\mathfrak{L} \models_{t'} \alpha'$ and for every t'' such that $t \leq t'' < t'$, $\mathfrak{L} \models_{t''} \alpha$;

8. $\mathfrak{L} \models_t X\alpha$ iff $\mathfrak{L} \models_{t+1} \alpha$

9. $\mathfrak{L} \models_t H\alpha$ iff for all $t' \leq t$, $\mathfrak{L} \models_{t'} \alpha$

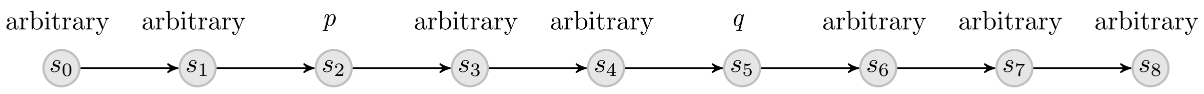
10. $\mathfrak{L} \models_t F^{-1}\alpha$ iff for some $t' \leq t$, $\mathfrak{L} \models_{t'} \alpha$

11. $\mathfrak{L} \models_t S(\alpha', \alpha)$ iff some $t' \leq t$ such that $\mathfrak{L} \models_{t'} \alpha'$ and for every t'' such that $t' < t'' \leq t$, $\mathfrak{L} \models_{t''} \alpha$;

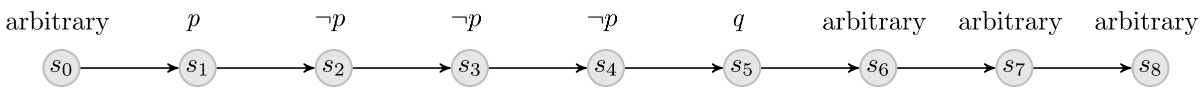
12. $\mathfrak{L} \models_t X^{-1}\alpha$ iff $\mathfrak{L} \models_{t-1} \alpha$

Let us consider a few examples of how our LTL can express properties. We provide LTL formulas with their intuitive semantics in a timeline diagram.

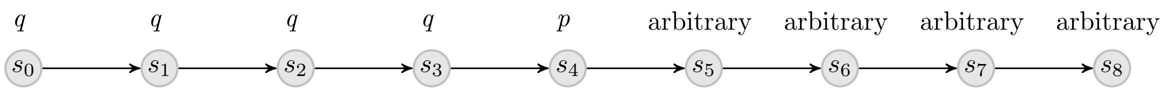
- $G(p \to Fq)$: q holds at some time after p holds.



- $G(p \to X(U(q, \neg p)))$: After p holds, it must not occur again until q holds.



- $G(p \to Hq)$: If p holds, q must hold at all previous states.



- $G(p \to F^{-1}q)$: Before p holds, q must hold.

$$\begin{tikzcd}
\text{arbitrary} & \text{arbitrary} & q & \text{arbitrary} & \text{arbitrary} & \text{arbitrary} & p & \text{arbitrary} & \text{arbitrary} \\
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] & s_3 \arrow[r] & s_4 \arrow[r] & s_5 \arrow[r] & s_6 \arrow[r] & s_7 \arrow[r] & s_8
\end{tikzcd}$$

- $G(p \rightarrow X^{-1}(X^{-1}q))$: If p holds, q must hold two steps earlier.

$$\begin{tikzcd}
\text{arbitrary} & \text{arbitrary} & \text{arbitrary} & q & \text{arbitrary} & p & \text{arbitrary} & \text{arbitrary} & \text{arbitrary} \\
s_0 \arrow[r] & s_1 \arrow[r] & s_2 \arrow[r] & s_3 \arrow[r] & s_4 \arrow[r] & s_5 \arrow[r] & s_6 \arrow[r] & s_7 \arrow[r] & s_8
\end{tikzcd}$$

In the above examples, each diagram gives only one way for the formula to hold, there being other possible diagrams which can be used to describe different views of the same formula.

Note that the temporal operators G, F, U, and X express properties in the future states while the temporal operators H ,F^{-1}, S, and X^{-1} express the same properties for the past instead of the future. We will denote LTL with the future operators only by LTL^{+}, to distinguish it from the full logic LTL.

2.4 Natural Language Specifications

The purpose of this section is to show what sort of sentences we are dealing with. The sentences that we refer to are written for specifying protocol requirements in formal specifications. Here are some typical sentences:

(1) Awready becomes low;

(2) Awvalid remains high;

(3) Awid is asserted.

We need to make certain assumptions about these sentences. For example, the above sentences contain various signal names (such as Awready, Awvalid and Awid) and these signal names are all treated as proper nouns. Based on their definitions in (ARM Ltd, 2012b), Awready stands for “write address ready”, Awvalid stands for “write address valid”, and Awid stands for “write address ID”. These signals are treated as time-dependent values since they have states where they can be either true or false.

There are certain other words in these sentences that must treated differently than the way they widely used in English. For example, the verbs such as “become”, “assert” and “remain” have special meanings in natural language specifications. Let us see what their interpretations are according the context they appear in. First of all, sentence (1) entails that the associated boolean variable “Awready” is false at a particular time during the evaluation because the verb “become” denotes as an event. Sentence (2) entails that the associated boolean variable “Awvalid” is true for several time points during the evaluation because the verb “remain” denotes a state. Sentence (3) entails that the associated boolean variable “Awid” holds at a particular time during the evaluation because the verb “assert” denotes an event. The distinction between events and states will be explained more in Section 3.2.

Let us turn to some words that are treated as temporal nouns. Consider the following sentences:

(4) Awready becomes low after every burst;

(5) Awvalid remains high until the EXOKAY response;

(6) Awid is asserted within MaxWaits cycles.

where “burst”, “response” and “cycles” here are temporal nouns. Based on their descriptions in ARM Ltd (2012b), a burst consists of a specified number of transfers. The “EXOKAY response” indicates that the associated transaction has passed. MaxWaits is a maximum number of cycles between particular two events. We treat MaxWaits as a constant. (Its value is actually 16.)

These temporal nouns denote particular events in the system where their main clauses “Awready becomes low”, “Awvalid remains high, and “Awid is asserted” must be evaluated with respect to the events present in those temporal nouns. Thus, these temporal nouns must be treated differently from the non-temporal nouns when we extract their meanings from natural language specifications (Section 4.5 shows how we make these distinctions). The entire list of all these assumptions is given in Appendix A.

2.5 SystemVerilog Assertions (SVA)

SVA is a subset of SystemVerilog which combines hardware descriptions and formal verifications. Assertions formally define the conditions required for an implementation to be correct. *SVA* has the capacity to define sequential expressions with clear temporal relationships between them. SystemVerilog has two types of assertions: immediate assertions and concurrent assertions. Immediate assertions are based on simulation event semantics. Moreover, they are executed in a similar way to an *if* statement in traditional programming languages. Obviously, they are not temporal in nature since they describe behaviours at instants of time. In contrast, concurrent assertions are primarily based on clock semantics. They use sampled values of variables. The sampled values are values of the variables used in evaluation. The assertions of benefit to us are of the concurrent type since they detect behaviours over periods of time. For example, assertion (7) checks that the signal a is high at every rising clock edge. If the signal a is not high at any rising clock edge, the assertion will fail. When the signal a is high, the value of it is “1’b1”.

(7) sequence Seq_1;
    @(posedge clk) a;
    endsequence.

Note that *SVA* syntax defines a sequence using a keyword pair “sequence-endsequence” with an associated name. Figure 2.5.1 shows a sample waveform for the signal a and how sequence Seq_1 reacts to this signal during simulation. The signal a goes to 0 on the positive edge of clock cycles 4 and 10, and hence the sequence fails. By contrast, the signal a goes to 1 on the rest of clock cycles, and hence the sequence succeeds.

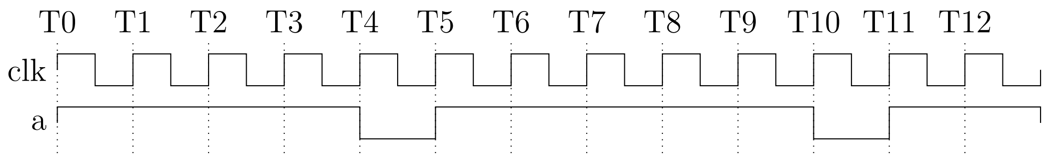


Figure 2.5.1: Waveform for sequence Seq_1 in (7).

SVA has a rich set of unary and binary temporal operators. Let us recall some of these operators from IEEE Std 1800-2012 (2013). Suppose we want to write assertions that check the following sequences of events:

(8) Awid must be asserted 4 clocks after Awready is de-asserted

(9) Once Awid is asserted, Awvalid is asserted two clocks later.

The corresponding assertions of the above sentences respectively will be as follows:

(10) sequence Seq_2;

@(posedge clk) not *AWREADY* ##4 *AWID*;

endsequence.

(11) sequence Seq_3;

@(posedge clk) *AWID* ##2 *AWVALID*;

endsequence.

In *SVA*, The ## operator defines delays in term of clock cycles - for instance, ##4 means a delay of 4 clock cycles as shown in assertion (10) between the signals not *AWREADY* and *AWID*. In *SVA*, We may use of one sequence inside another as shown in (12).

(12) sequence Seq_4;

@(posedge clk) Seq_2 ##2 *AWVALID*;

endsequence.

Assertion (12) is written to describe sequences (10) and (11) together.

Another important use of the ## operator is that we can specify a range of absolute delays by it. For example,

(13) *AWID* ##[1:4] *AWVALID.*

Assertion (13) states that, when the signal *AWID* becomes true, the occurrence of the variable *AWVALID* could be delayed from one cycle up to 3 further clock cycles. Note that we can specify an assertion with infinite repetition range such as in (14) where the $ symbol indicates that the signal *AWVALID* will eventually occur.

(14) *AWID* ##[1:$] *AWVALID.*

Now we will discuss how we specify a signal or sequence that occurs a number of times. In *SVA*, we can apply consecutive repetition operator [*] to a sequence to indicate that the sequence repeats itself either a fixed number of times or presents a range of repetitions. If a sequence repeats in a range of repetition from n to m, then maximum of m times can be specified as '$', which indicates a finite but unbounded range.

Let us illustrate how we use consecutive repetition operator [*] in SVA. For example, assertion (15) states that the signal *AWID* is true, and must remain true for exactly 2 further clock cycles, 2 clock cycles before which the variable *AWVALID* will be true.

(15) *AWID*[*3] ##2 *AWVALID*;

this is equivalent to

(16) *AWID* ##1 *AWID* ##1 *AWID* ##2 *AWVALID.*

Using the repetition operator [*] allows us to express sequences in a range of repetition such as the following assertion:

(17) *AWID*[*1:3] ##2 *AWVALID*,

which asserts that, the signal *AWID* is true, and may remain true for up to 2 further clock cycles, 2 clock cycles after which the signal *AWVALID* will be true. Moreover, we may write assertion (17) in another way as follows:

(18) *AWID* ##2 *AWVALID*
or *AWID* ##1 *AWID* ##2 *AWVALID*
or *AWID* ##1 *AWID* ##1 *AWID* ##2 *AWVALID*,

where the resultant of the "or" operands is true whenever at least one of the sequences is true. The "or" and "and" operators work with at least two sequences or Boolean expressions. The "and" operator means that if both sequences or Boolean expressions are true, then the result of "and" operation is true. Note that if the maximum number of a range is unknown, then we express that by using the form a[*1:$] which means the expression a will stands for finite, but unbounded number of iterations using '$' which indicates the maximum limit of the range as mentioned earlier. For example, we can write an assertion using this form as follows:

(19) *AWID*[*1:$] ##2 *AWVALID*,

which asserts that, the signal *AWID* is true, and must remain true until 2 clock cycles before the variable *AWVALID* will be true.

Let us review some more *SVA* operators. There are two types of implication in *SVA*: an overlapped implication and a non-overlapped implication. The overlapped implication is denoted by the symbol |–>, while the non-overlapped implication is denoted by the symbol |=> as shown in the following examples, respectively.

(20) *AWID* |–> *AWVALID*

(21) *AWID* |=> *AWVALID.*

Assertion (20) states that, if the antecedent *AWID* holds, then the consequent expression *AWVALID* holds in the same clock cycle. On the other hand, assertion (21) states that, if the antecedent *AWID* holds, then the consequent expression *AWVALID* holds in the next clock cycle. Note that assertion (21) can be expressed differently using the "##" operator as shown below in (22), where ##1 means a delay of one clock cycle before the consequent expression *AWVALID* holds.

(22) *AWID* |–> ##1 *AWVALID.*

Now let us discuss important operators in *SVA* called sequence match operators. These operators are: *throughout*, *within*, *ended* and *first_match.* These operators take sequences as operands and generates new sequences as a result. Moreover, these operators can take sampled values of expressions and generate true or false as a result. The syntax of these operators is shown below:

(expression) throughout (sequence);

(sequence) within (sequence);

(sequence).ended;

first_ match(sequence).

The *throughout* operator is used to assert that a certain expression is valid over the period of the sequence. Meanwhile, the *within* operator is used to assert that there is the containment of one sequence within another sequence. For example, the following assertions illustrate how the *throughout* and *within* operators are used in practice:

(23) *AWID* throughout Seq_4;

(24) Seq_1 within Seq_4.

Assertion (23) asserts that, the signal *AWID* must be true at every clock cycle during the occurrence of the sequence Seq_4. By contrast, assertion (24) asserts that, the sequence Seq_1 happens within the start and completion of the sequence Seq_4.

The *ended* operator returns a Boolean value true or false depending on whether the associated sequence achieves a match on that particular clock cycle. Consider the following example:

(25) Seq_1 ##1 Seq_3.ended,

which asserts that, the sequence Seq_3 must be completed one clock cycle after the sequence Seq_1 completes (regardless of when Seq_3 starts). Figure 2.5.2 shows a sample waveform for assertion (25) and how sequence Seq_3 terminates one clock cycle after the sequence Seq_1 terminates during simulation.

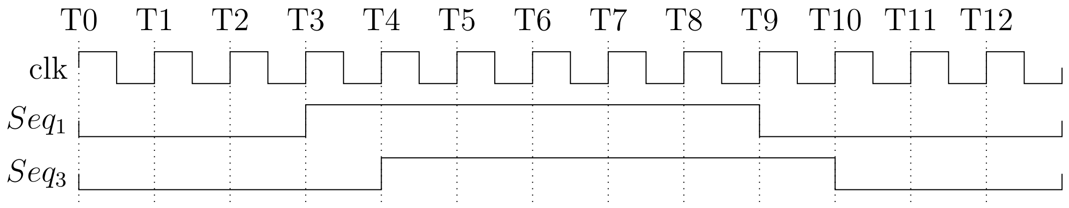


Figure 2.5.2: Waveform shows a successful match of assertion (25).

The *first_match* operator matches only the first occurrence of its sequence argument and discards all subsequent matches from consideration. For example,

(26) first_match(*AWID* ##[1:2] *AWVALID*),

whichever of the (*AWID* ##1 *AWVALID*) and (*AWID* ##2 *AWVALID*) matches first becomes the result of the *first_match* operator.

In *SVA*, there are built-in functions that help in accessing the sampled values of an expression or detecting changes in a sample value of an expression. These functions are **$rose**, **$fell**, **$stable** and **$past**. The **$rose**, **$fell** and **$stable** functions compare the values of their arguments in the present cycle with the previous cycle. The **$past** function returns the previous value of its argument. Table 2.5.1 shows the syntax of these functions.

Functions
$rose(Boolean expression or signal name)
$fell(Boolean expression or signal name)
$stable(Boolean expression or signal name)
$past(signal name, number of clock cycles)

Table 2.5.1: The built-in functions with their syntax.

The **$rose** function evaluates to true if the value of the expression is changed to 1. On the other hand, the **$fell** function has the inverse behaviour, and evaluates to true if the least significant bit of the expression changed to 0. The **$stable** function evaluates to true if the value of the expression did not change. The **$past** function gets the value of its signal from the previous clock cycle. The number of clock cycles in the past can be given using the second parameter.

Lastly, SVA has the standard LTL operators *X*, *G*, *F* and *U* but with the different names as follows:

$$\begin{aligned} \mathrm{X} &\equiv \text{nexttime} \\ \mathrm{G} &\equiv \text{always} \\ \mathrm{F} &\equiv \text{eventually} \\ \mathrm{U} &\equiv \text{s\_until} \end{aligned}$$

The *LTL* operators are defined in Section 2.3. Thus, we skip discussing them here to avoid repetition. All of the above-mentioned operators will be used for constructing transformation rules between TPL and *SVA*. For a more detailed account of *SVA* refer to IEEE Std 1800-2012 (2013).

2.6 Interval Temporal Logic

In this section, we introduce *interval temporal logic* (henceforth, *ITL*). We recall the semantics of *ITL* presented in Pratt-Hartmann (2005). ITL is a first-order language having variables which range over time-intervals, and predicates corresponding to event-types and temporal order-relations. Pratt-Hartmann (2005) follows the same way of introducing modal operators as in Halpern and Shoham (1991).

Let us recall the formal definitions of *ITL* which will be used throughout the thesis.

Definition 2.6.1. In the sequel, let $\mathcal{I}_{\mathcal{R}}$ be a set of intervals, where an interval is a closed, bounded, non-empty subset of the real numbers. Moreover, we use the letters *I*, *J*, ... with or without decorations as temporal variables ranging over $\mathcal{I}_{\mathcal{R}}$.

Definition 2.6.2. Let I = [a,b] and J = [c,d] be intervals. If a < c < d < b, we let the terms *init(J,I)* and *fin(J,I)* denote the intervals [a,c] and [d,b], respectively, where *init* and *fin* are partial functions to denote the initial segment of I up to the beginning of *J*, and the final segment of I from the end of *J*, respectively.

Let us review the semantics of ITL using examples from natural language specifications. Consider the following sentences:

(27) *Awid* is asserted

(28) *Awid* is asserted during every cycle.

Sentence (27) states that, within some contextually specified interval of interest, there is an interval over which *Awid* is asserted. The meaning of (27) can be represented as follows:

(29) $\exists J_0(assert(Awid, J_0) \wedge J_0 \subseteq I)$.

Note that the quantification of J_0 is limited to the temporal context which is represented by the free variable *I*. We can assume that the value of this variable is assigned by the context of utterance. For example, I may denote a time interval over which some system is required to have the specified behaviour.

Sentence (28) states that, within the temporal context, every interval over which a cycle occurs includes an interval over which *Awid* is asserted. The meaning of (28) can be represented as follows:

(30) $\forall J_1(cycle(J_1) \wedge J_1 \subseteq I \rightarrow \exists J_0(assert(Awid, J_0) \wedge J_0 \subseteq J_1))$.

The noun *cycle* is considered in Pratt and Francez (2001) as a temporal noun which denotes an interval time such as *meeting*, *Monday* and *1995*. Thus, the meaning of *cycle* should be as follows:

(31) $\lambda J \lambda I.cycle(J) \wedge J \subseteq I$,

Intuitively, the word "cycle" picks out those intervals J over which a cycle occurs with some temporal context I as shown in formula (30).

Let us now turn to events that take place not during, but before or after various time intervals. In ITL, we use *init* and *fin* functions, described in Definition 2.6.2, to express the *before* and *after* relations, respectively as shown in Figure 2.6.1

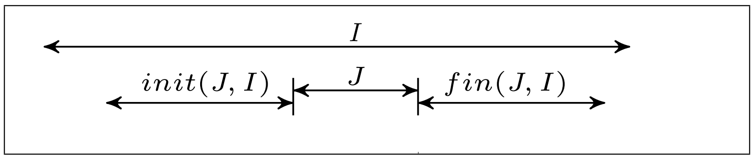


Figure 2.6.1: Illustrating the functions $init(J, I)$ and $fin(J, I)$.

where *init(J,I)* denotes the initial segment of I up to the beginning of *J*, while *fin(J,I)* denotes the final segment of I from the end of *J*.

Let us consider a few examples of how we interpret English sentences having the *before* and *after* relations into ITL. Consider the following sentences:

(32) *Awid* is asserted during every cycle until *Awvalid* goes high

(33) After the response phase, *Awid* must be asserted during every cycle until *Awvalid* goes high.

The semantics of sentences (32) and (33) in ITL are as follows, respectively

(34) $\imath J_2(high(Awvalid, J_2) \wedge J_2 \subseteq I,$
$\quad \forall J_1(cycle(J_1) \wedge J_1 \subseteq init(J_2, I) \rightarrow \exists J_0(assert(Awid, J_0) \wedge J_0 \subseteq J_1))).$

(35) $\imath J_3(response\_phase(J_3) \wedge J_3 \subseteq I,$
$\quad \imath J_2(high(Awvalid, J_2) \wedge J_2 \subseteq fin(J_3, I),$
$\quad\quad \forall J_1(cycle(J_1) \wedge J_1 \subseteq init(J_2, fin(J_3, I)) \rightarrow$
$\quad\quad\quad \exists J_0(assert(Awid, J_0) \wedge J_0 \subseteq J_1)))).$

In these examples, we employ the definite quantifier $\imath x(\psi, \psi')$ with the standard (Russellian) semantics. Formula (34) states that, within the temporal context I, there is a unique interval J_2 such that *Awvalid* goes high at J_2 and every interval J_1 such that J_1 is a cycle and J_1 is contained in $init(J_2, I)$, in turn includes an interval J_0 over which *Awid* is asserted. Formula (35) asserts that, within the temporal context I, there is a unique interval J_3 such that the response phase occurs at J_3, and there is a unique J_2 subinterval of $fin(J_3, I)$ such that *Awvalid* goes high at J_2 and every interval J_1 over which a cycle occurs includes an interval J_0 over which *Awid* is asserted, and moreover J_1 is contained in $init(J_2, fin(J_3, I))$.

Generally speaking, ITL is a suitable logical form for encoding events and their temporal locations from English sentences. In this section, we have given a flavour of the language ITL. For a complete specification, we refer the reader to Pratt-Hartmann (2005).

In this study, we capture the semantics of temporal constructions from natural language specifications using *ITL* which is known as *interval-based logic*. Then, we generate *SVA* (which is essentially *point-based logic*) from *ITL*. *SVA* has different characters than *ITL*.

Can we connect them? Can we map from English to SVA either via TPL translations or directly? These questions are addressed in Chapter 5.

Chapter 3

Linguistic Background

The aim of this chapter is to provide an overview of the field of temporal semantics from the point of view of linguistics. It begins with semantic theories of events. Following is an overview of grammatical aspect and aspectual class of verb phrases. Finally, this chapter reviews the semantic features of temporal prepositions as well as their formal representations.

3.1 Theory of Events

Semantic theories using quantification of implicit variables ranging over events have been applied to a variety of problems in linguistic theory such as the semantics of temporal prepositions, perceptions, thematic relations, and nominalisation. These theories, however, differ from each other in respect of the variables they employ and the way in which these variables are added to the non-logical predicates involved. Event semantics refer to semantic analyses which adopt the suggestion of Davidson (1967) that specific predicates pick an implicit variable over events as an argument. In Davidson's initial proposal, this event argument is included as an additional argument to the predicate. The event variable is existentially quantified, with the result that sentence (36) is assigned a logical form such as (37) instead of the classical treatment as in (38).

(36) Brutus stabbed Caesar;

(37) $\exists e$(stab(Brutus, Caesar, e));

(38) stab(Brutus,Caesar).

Thus *stab* is analysed as expressing a three-place relation, between the person who stabs, the person who gets stabbed, and a stabbing event; and the sentence is analysed as asserting that, there exists an event in which Caesar is being stabbed by Brutus.

The primary motivation for this solution is that it offers a way to analyse prepositions such as temporal, spatial and instrumental preposition phrases. These prepositions can be treated as predicates of the event or more particularly as predicates that share the event variable of the verb. Every preposition takes a two-place predicate over an event, which applies to the verbal event argument using an ordinary propositional conjunction. The existential quantification that binds the event variable takes scope over the whole sentence structure. For example sentence (39) is assigned to the logical structure (40).

(39) Brutus stabbed Caesar in the back with a knife.

(40) $\exists e$(stab(Brutus, Caesar, e) $\wedge$ in(e, the back) $\wedge$ with(e, a knife)).

Note that the predicates *stab*, *in*, and *with* are linked because they apply to the same event. This approach offers an elegant solution to the problem that arises whereby prepositions are treated as arguments of the verb. Consequently, we avoid expressing *stab* in a four-place relation as in (41).

(41) stab(Brutus, Caesar, the back, a knife).

If we adopt a logical structure such as (41) and continue to express Brutus stabbed Caesar as in (38) with a two-place relation, we cannot claim that what *stab* expresses has the same meaning in both sentences; on the contrary, we would say it is ambiguous. Therefore, adopting this approach will cause massive ambiguity when we express sentences that have the same predicate with more or fewer prepositions. For instance, *stab* will be expressed with different relations in each of the sentences below:

(42) Brutus stabbed Caesar in the back with a knife.

(43) Brutus stabbed Caesar in the back.

(44) Brutus stabbed Caesar with a knife.

(45) Brutus stabbed Caesar.

Sentences with more prepositions typically entail the same sentences with fewer prepositions as stated in Bartsch and Kiefer (1976) and Davidson (1967). Hence, using this analysis, we correctly capture the fact that sentence (42) entails all the other sentences (43)–(45), where sentence (42) also entails the conjunction of sentence (43) and sentence (44) but not inversely, because the conjunction of (43) and (44) does not require that the two stabbings are the same. Moreover, sentence (43) or sentence (44) can entail sentence (45), however sentence (45) cannot entail sentence (43) or sentence (44).

Another advantage of this approach is that certain types of nominalisation require an analysis in terms of events. Thus, Higginbotham (1985) and Parsons (1990) express event nominals with the same predicate of events as the verbs in which event nominals can be explicitly quantified over the event. Adopting this suggestion provide a nice account for the validity of the argument. Consider the following examples:

(46) In every burning, oxygen is consumed;

(47) Agatha burned some wood;

(48) Oxygen was consumed.

In traditional logic, inferring sentence (46) from sentences (47) and (48) is problematic, since the burning event in sentence (46) is not expressed in the same way as the individual events of the verbs *burned* and *consumed.* Applying explicit quantification over event nominals by using variable reference to events allows us to find inferences and connect it with explicit quantification over events such as the logical relational between sentence (46) and sentences (47) and (48) in a very straightforward manner. For example, Parsons (1990) shows how the quantification over an event burning in (46) can be logically related to (47) and (48). The logical forms that associate sequentially with that are:

(49) $\forall e(burn(e) \rightarrow \exists e'(consume(e') \land object(e', oxygen) \land in(e, e')))$.

(50) $\exists e(burn(e) \land subject(e, Agatha) \land object(e, wood))$.

(51) $\exists e'(consume(e') \land object(e', oxygen))$.

where the burning event in sentence (46) is formally represented using the same type of event as the burning referred to in sentence (47). As a result, sentence (48) follows from (46) and (47), since the logical form of (48) can be derived by using the resolution inference rules which follows from the logical form of (46) and (47).

Event semantics has also been successfully applied to a variety of other problems, although a discussion of them here is not necessary for the purpose of this thesis: perceptions (Higginbotham, 1983; Vlach, 1983; Parsons, 1990), adjectives (Larson, 1998), temporal anaphora and narrative progression (Hinrichs, 1986; Partes, 1984), plurality (Schein, 1993; Landman, 2001), and many other phenomena.

3.2 Grammatical Aspect and Aspectual class

In this section, we give an overview of grammatical aspect and aspectual class of verbs. According to Comrie (1976), aspect is concerned with different ways of describing the internal structure of events based on the sentences or discourses in which they appear. Lehman (1972) calls aspect the status of a situation. Within the study of aspect, there are two types of aspectual distinction in language: grammatical and lexical aspects.

Grammatical aspect refers to what has traditionally been termed “aspect” (Comrie, 1976), “aspectual form” (Dowty, 1979) and “viewpoint aspect” (Smith, 1991). Grammatical aspect expresses how an event or state extends over time. In English, grammatical aspect has two different dimensions: perfective/non-perfective and progressive/non-progressive. There are 4 possible combinations of these forms as shown in Table 3.2.1. Each aspect realizes by a specific marker. Note that the $\emptyset$ symbol means no marker between non-perfective and non-progressive aspect forms.

Aspect	Combination of	Marker
Simple	Non-perfective and Non-progressive	$\emptyset$
Perfective	Perfective and Non-progressive	Have + -en
Progressive	Progressive and Non-perfective	Be + -ing
Progressive Perfective	Progressive and Perfective	Have been + -ing

Table 3.2.1: Grammatical aspect types and their Markers in English

These aspects can be illustrated for the verb “build” in (52) – (55) where they locate the situations in the past. Note that these aspects can be available in all tenses.

(52) John built a house.

(53) John has built a house.

(54) John was building a house.

(55) John has been building a house.

The simple aspect (or indefinite aspect as it is more frequently called) is a combination of the non-perfective and non-progressive aspects. Any verb (in the past, present or future) in the simple tense is consider to be in the simple aspect. The simple aspect does not tell us whether the event is a complete event or a habitual event. Any verb in the simple aspect depends on the context and tense to determine whether the event is complete or in progress. Let us illustrate this by giving one more sentence as follows:

(56) John built houses.

The "building" event in (52) is a complete event because the tense and context tells us this event is not a habitual event. Moreover, the semantic nature of the verb "build" tells us the event has a culmination point (this will be discussed later in the lexical aspect domain). However, the "building" event in (56) is unclear whether this is a habitual event or not based on the context; even though the tense tell us this event is a complete event.

The perfective aspect is formed with the auxiliary verb *have* and the past (-en) participle. The auxiliary verb *have* can be changed to *has*, *had*, or added by the future marker *will.* The perfective aspect indicates that the event is to be viewed as a bounded whole, and looks at the event from the outside, without reference to any of its internal structure. For example, the verb *built* in (53) reports the event as complete, with both initial and terminal points. Thus, sentence (53) implies that there is a house and that John built it.

The progressive aspect is formed with the auxiliary verb 'to be' and the lexical verb in the the present (-ing) participle form. The progressive aspect looks at the event from inside, or looks inside its internal temporal structure. For example, the verb in (54) reports the event as in progress, and does not reference to its initial and terminal points. Thus, the meaning of sentence (54) does not tell us whether or not John has completed building the house.

The progressive perfective aspect is a combination of the progressive and perfective aspects. The perfective progressive aspect is formed with *have been* and the lexical verb in the present (-ing) participle form. The progressive perfective aspect expresses an ongoing situation that began somewhere in the past but that is still incomplete at the reference time. For example, the verb "has been building" in (55) reports the event that stretches backward from now and continues into the present moment. There is also the sense of incompleteness since the event in progressive form which does not tell us whether or not

John has finished building the house.

Now, let us begin to discuss lexical aspect. Lexical aspect refers to what has been termed "Aktionsart", but also "aspect" (Verkuyl, 1972), "inherent lexical aspect" (Comrie, 1976), "aspectual class" (Dowty, 1979), "situation aspect" (Smith, 1991) and "eventuality type" (Bach, 1986; Filip, 1999). Lexical aspect denotes the internal temporal structure of an event, which is determined by the semantic nature of the verb, by the properties of the verb's argument, and by the way the verb is related to its argument. Much work on lexical aspect depends on the aspectual classes initially introduced by Vendler (1967). Vendler identified four aspectual verb classes based on static vs. dynamic, punctual vs. durative, and telic vs. atelic. In the Vendler classification, verbs may denote states, activities, accomplishments or achievements.

State and activity verbs have atelic situation aspect (unbounded). State verbs denote situations that last for certain periods of time with non-dynamic (static) changes throughout their entire duration. By contrast, activity verbs denote dynamic situations that occur over a duration of time.

denotes several instances of the action

For example, consider the following sentences:

(57) John liked Mary

(58) John ran.

In sentence (57), the verb "liked" is a state verb which denotes a non-dynamic situation, whereas in sentence (58), the verb "ran" is an activity verb which denotes a dynamic situation. Both situations extend in time and have no culmination point.

Accomplishment and achievement verbs have dynamic and telic situation aspects (bounded). Accomplishment verbs denote events that take place over a duration of time.

On other hand, achievement verbs denote events that occur instantaneously – they are inherently punctual in nature– at a specific point in time. Consider the following examples:

(59) Mary wrote a letter

(60) Chris reached the summit.

In sentence (59), "wrote a letter" is an accomplishment event that extends in time but it has a culmination point. By contrast, "reached the summit" in sentence (60) is an achievement event that occurs at instantaneous time and has a culmination point.

Dowty (1979) proposed several diagnostic tests to distinguish one class from other classes. Dowty combines aspect markers and temporal prepositions with verbs to decide whether verbs have different interpretations or not after having restrictions by these markers and prepositions. Thus, he proposed a list of different verbs or verb phrases that correspond to each aspectual class as instances of Vendler's four classes, and the list is presented in Table 3.2.2.

States	Activities	Accomplishments	Achievements
know	run	run a mile	recognize
believe	push a cart	draw a circle	reach the summit
think	breathe	build a house	die
like	write	write a letter	arrive

Table 3.2.2: List of examples of verbs or verb phrases and their assignment to aspectual classes based on Dowty (1979).

Several other classification systems have been proposed since Vendler's work. Some of this work makes the same distinctions, dividing some classes or removing others. We intend to describe the most relevant contributions for distinguishing aspectual classes. In particular, we show the classification systems for aspectual classes developed by Bach (1986) and Moens and Steedman (1988). Let us begin with Bach's work.

Bach (1986) adopts the term 'eventualities' for aspectual classes. His primary proposal for distinguishing aspectual classes is presented in Figure 3.2.1.

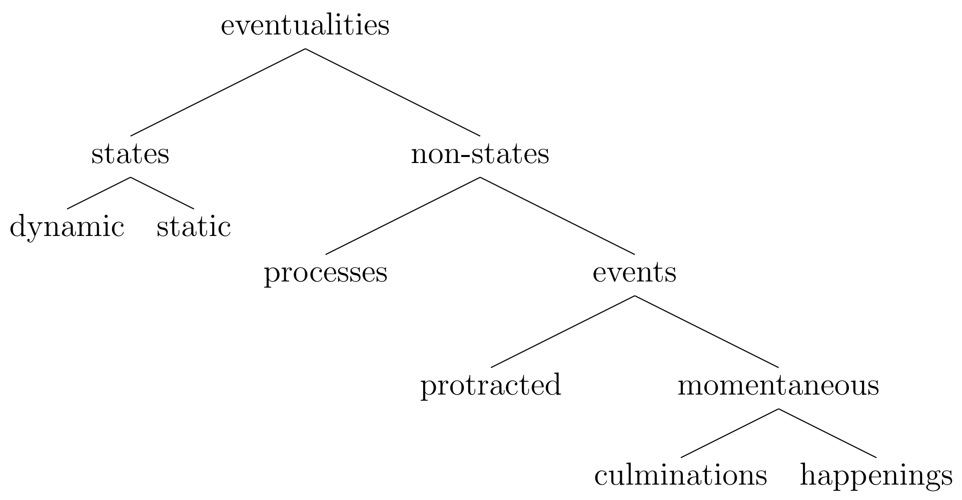


Figure 3.2.1: Classification of aspectual classes in Bach (1986)

As shown in Figure 3.2.1, eventualities have divided into states and non-states. States have two types of state verbs based on their ability to occur with the progressive: static states and dynamic states. Static states (equivalent to Vendler's states) do not occur with progressive as shown in sentence (61). On other hand, the dynamic state verbs can freely occur with progressive as shown in sentence (62). Dynamic states are episodic in the sense that they only apply to 'spatio-temporal slices' of individuals. Bach's distinction on states mainly follows Carlson's (1981) analysis.

(61) * John is knowing the answer. (static state)

(62) Mary is feeling cold. (dynamic state)

Non-states are divided into processes (equivalent to Vendler's activities) and events (including Vendler's accomplishments and achievements). Events are either protracted (equivalent to Vendler's accomplishments) or momentaneous (equivalent to Vendler's achievements). Momentaneous events are further subdivided into culminations (e.g. die, win, reach the top) and happenings (e.g. knock, notice, recognize, and flash once). Happenings normally occur very quickly, with no result other than the occurrence of the event. Thus, they are characterised by the dynamic, atelic and punctual features such as in sentence (63), but culminations involve a transition to a new state that is associated with telic features such as in sentence (64).

(63) Mary knocked at the door. (happening)

(64) John won the race. (culmination)

Moens and Steedman (1988) extend Vendler's classification and introduce another aspectual class called points that have punctual and atelic feature as with the happening events in Bach's classification. Point classes have no consequences and also have virtually no time such as "cough","tap" and "wink". The authors have changed the Vendler's classification to different names, for example activities to processes, accomplishments to culminated processes, and achievements to culminations. They distinguish five aspectual types: states, processes, culminated processes, points and culminations as shown in Table 3.2.3.

	Event		**States**
+conseq	atomic	extended	understand, love, know, resemble
	Culmination recognize, spot, win the race	**Culminated Process** build a house, eat a sandwich	
-conseq	**Point** hiccup, tap, wink	**Process** run,swim, walk, play the piano	

Table 3.2.3: Classification of aspectual classes in Moens and Steedman (1988).

As shown in Table 3.2.3, they give a clear distinction between states and all the other categories, denoted "events". States now include only stative verbs that make no time-related reference to start and end points. For instance, a sentence such as "John is tall" does not refer to a certain point in time at which the state of John being tall began (unbounded). Within the event categories, they delimit these classes based on atomic vs. extended events, and +consequent state vs. -consequent state. A consequent state is often considered to coincide with a telic feature. For example, a sentence such as "John ran a mile" has an accomplishment event which occurs over an extension of time but it has a telic situation aspect (bounded). Therefore, the consequence of John running a mile takes place and it can be referred to by using a perfect entailment to denote that John has run a mile. Conversely, there are events that have a period of time without a telic situation aspect. Thus, they do not refer to consequences such as activities in a sentence like "John plays the piano" which has no consequence of playing the piano or that has a particular finishing point that could be associated with a consequence of the event.

We have seen all these different aspectual classes in natural language. We need to be able to process them grammatically. We also know that they have various semantic effects. What do we require for our problem? Probably, for the purpose for handling natural language protocol specifications, we only need to distinguish between events and states, and here is why. Consider the following examples:

(65) Awid remains low until Awvalid occurs.

(66) * Awid goes high until Awvalid occurs.

The verb "remains" in (65) is a state verb which indicates that the situation has no culmination point and extends in time. Therefore, we assign universal quantification to the verb "remains". The meaning of *until* preposition requires its main clause to be restricted with universal quantifier (this restriction is discussed further in the next section). Therefore, we can say sentence (65) is acceptable while (66) is not acceptable since the verb "goes" is an event verb which takes existential quantification. Thus, we need to build a semantic grammar that can only accept the correct sentences. Section 4.5 describes how we build our semantic grammar to enforce these assumptions. In Section 4.3.2, we show how we encode aspetucal classes, namely events and states, and grammatical aspects for our problem.

This section has explained the central importance of aspectual classification in natural language semantics. The purpose of the current study was to determine aspectual categories for verbs and sentences in the contexts. We have shown the classification systems for aspectual classes that developed by Vendler (1967), Bach (1986) and Moens and Steedman (1988). However, in this thesis, the only thing we really require is to make a distinction between states and events.

3.3 Temporal Prepositions

Temporal adverbials report temporal relations between a pair of times or events. These include adverbs (e.g. yesterday), noun phrases (e.g. last year) or prepositional phrases (e.g. on Monday). In English, we can express time-related information in terms of point, interval, frequency and temporal relationship. When expressing a time point, temporal prepositions can precisely locate the exact time an event occurs, as in sentence (67). When expressing a time interval, temporal adverbials can denote the event occurs within a period of time, as in sentence (68). Frequency adverbials denote how often an event occurs, as in the temporal noun "every Tuesday" in (69). Temporal relationships –often introduced by *after*, *before*, *during*, etc.– are used to state that an event described in the main clause may be next to, previous to, or concurrent with the time of the event described by the temporal clause, as in sentence (70).

(67) Mary arrived home at 1:30 pm.

(68) John was in Boston last year.

(69) Mary ate a pizza every Tuesday.

(70) Mary visited France after she finished her book.

In this section, we present an account of English sentences featuring temporal prepositions. Then, we show the effects of the interaction of temporal prepositions with temporal temporal expressions (aspectual classes, grammatical aspects and tense) on the temporal ordering of events and states.

3.3.1 The Semantics of Temporal Prepositions

Adequate representations for the semantics of temporal prepositions are provided by a variety of formal approaches (see Bennett, 1975; Kamp, 1981; Brée and Smit, 1986; Richards et al., 1989; Allen and Ferguson, 1994). However, our focus will be on approaches that we think can improve on the practice of, for example, Pratt and Brée (1993), Pratt and Francez (1997, 2001), and Pratt-Hartmann (2005). These other approaches fail to make use of their theory in practice for encoding the semantics of temporal prepositions systematically.

Let us begin by introducing the semantics of temporal prepositions based on interval-based logic. Consider the following sentences:

(71) Mary arrived home between 1 o'clock and 2 o'clock.

(72) John worked in Boston from 1996 until 2000.

Sentence (71) reports an event –namely, Mary's arriving home– and locates that event as having occurred during the interval [1:00, 2:00]. Sentence (72) also reports an event –namely, John's working in Boston– but locates that event as having occurred over the entire interval [1996, 2000].

Now let us express sentences (71) and (72) without tense markers as follows, respectively.

(73) Mary arrive home between 1 o'clock and 2 o'clock.

(74) John work in Boston from 1996 until 2000.

which receive truth-values with respect to intervals of time. Therefore, (73) is true over an interval J fully contained within the interval [1:00, 2:00], such that "Mary arrive home" is true over J; also, (74) is true over every interval J fully contained within the interval [1996, 2000], such that "John work in Boston" is true over J. We can write these truth-conditions using ITL (described in Section 2.6) as follows

(75) $\exists J(J \subseteq [1{:}00, 2{:}00] \wedge (\text{Mary arrive home})(J))$

(76) $\forall J(J \subseteq [1996, 2000] \rightarrow (\text{John work in Boston})(J))$

where tenseless sentences interpreted as one-place predicates have arguments which range over time intervals. There are alternative approaches to the semantics of tensed sentences such as those of Reichenbach (1947), Bäuerle and von Stechow (1980), Parsons (1989), and Blackburn (1994) who adopted logics based on time points. However, the framework adopted here performs well with most temporal constructions in English sentences.

We have performed the task of mapping the underlying tenseless sentences onto prepositions and establishing how temporal prepositions can play an important role for providing the semantics of tenseless sentences over intervals, as in (73) and (74). However, sentences (71) and (72) contain, beside prepositional phrases and past-tense markers, a grammatical aspect (simple, perfective, progressive and/or progressive perfective aspect) and an aspectual class (state, activity, accomplishment and/or achievement classes). These aspect-markers can also play a role in determining the truth-values of English sentences. To give an idea of how these aspect-markers are essential for providing proper interpretations, take for instance the verb "arrive" in sentence (71) which is an achievement verb, whereas "work" in sentence (72) is an activity verb. As explained in Section 3.2, an achievement class has punctual and telic features which naturally takes existential quantification over intervals. By contrast, an activity class has durative and atelic features which normally takes universal quantification over intervals. Therefore, we should take these factors into account when we specify the truth-values of English sentences.

Within this approach, the reference interval I is specified either indexically or anaphorically. [1] Let us give an example of an indexical reference with English sentences; consider the following examples:

(77) John has been living in France since 2014.

(78) John has been living in France for two years.

where the truth-conditions of (77) and (78) are plausibly given, respectively, by

(79) $\forall J(J \subseteq [2014, \text{TOU}] \rightarrow (\text{John live in France})(J))$.

(80) $\forall J(J \subseteq [\text{TOU - 2yrs, TOU}] \rightarrow (\text{John live in France})(J))$.

As the reference interval above is bounded by the time of utterance (henceforth TOU), we call this an *indexical usage.* The perfect tense in sentences (77) and (78) indicates the reference interval stretches backward from the TOU. In Pratt and Brée (1993), it is suggested that the temporal prepositions *since* and *for* restrict their main clause to be quantified universally over a time interval as shown in (79) and (80). In fact, the verb *live* is a state verb which has durative and atelic features. Thus, the tenseless sentence "John live in France" naturally takes universal quantification over a time interval. With respect to anaphorical cases for specifying the reference interval I, we refer the reader to Pratt and Brée (1993). Our interest here is to give an introduction of interpreting English sentences using interval-based semantics rather than to give a full specification.

Another approach to interval-based semantics for capturing English sentences featuring temporal prepositions is given by Pratt and Francez (1997, 2001). These proposals can adequately provide compositional semantics of temporal preposition phases, with special emphasis on their quantificational roles. Their theory is to use generalised temporal quantifiers to represent temporal noun phrases, temporal preposition phases, and sentences. The advantage of using their framework is that it provides an elegant account for cascaded

[1]In Pratt and Brée (1993), the reference interval I classified into the eight reference intervals with respect to the time of utterance (TOU) and the reference time (TOR).

temporal preposition phase modifications in English sentences, such as:

(81) John telephoned Mary during every meeting until Christmas.

These cascades can be handled correctly by their semantics. Their proposal leads to the following interpretation for sentence (81):

(82) $\imath J_2(Christmas(J_2) \wedge J_2 \subseteq I,$
$\quad\forall J_1(meeting(J_1) \wedge J_1 \subseteq init(J_2, I) \rightarrow$
$\quad\quad\exists J_0(\text{ (John telephone Mary)}(J_0) \wedge J_0 \subseteq J_1))).$

Before giving an explanation of the above formula, recall from Section 2.6 that the terms init(J,I) and fin(J,I) denote the initial segment of I up to the beginning of J, and the final segment of I from the end of J, respectively. The meaning of (82) asserts that, within the temporal context I, there is an interval of John's telephoning Mary that holds at every meeting until the unique interval of Christmas in the temporal context I. Notice that, we assume Christmas to have a missing determiner which contributes a definite article. The authors suggest that if there is no overt determiner in a temporal preposition phase, we need to assign an article to that temporal preposition phase based on the one that is most suitable semantically. Thus, we quantify the meaning of *until*'s complement with definite quantification, since *until* requires its complement to be definite quantified as shown in (83). This restriction is also applied to *by* as shown in (84), which *by*'s complement resists being with anything other than a definite article.

(83) John telephoned Mary every day until {the exam/* every exam/* an exam}.

(84) John telephoned Mary by {the exam/* every exam/* an exam}.

This discovery constitutes the second advantage of these proposals, which is restricting temporal preposition phrases with some particular quantifications to generate the correct truth-conditions of sentences featuring temporal prepositions. This provides adequate systematic treatments of the semantics of sentences with even multiple temporal preposition phrases.

Let us give an example of a sentential complement in a temporal preposition phrase based on Pratt and Francez's (2001) approach. Consider the following sentence:

(85) John telephoned Mary whenever she arrived.

where the temporal preposition *whenever* requires its complement sentence to be universally quantified. Thus, sentence (85) asserts that, an event of John's telephoning Mary occurs after every interval over which Mary's arriving. Note that not all temporal prepositions have restriction on their complements. For example temporal prepositions like *during*, *at* and *on* allow their complements to have any quantification pattern as in sentences (86) and (87).

(86) John telephoned Mary during {the exam/ every exam/ an exam}.

(87) John telephoned Mary on {Tuesday/ a Tuesday/ every Tuesday}.

Let us now turn to quantification restrictions on the modificand of temporal prepositions. Pratt and Francez (1997, 2001) observe that some temporal prepositions require their modificands to be universally quantified rather than existentially quantified such as *for*, *until* and *throughout* as shown in the following examples:

(88) John telephoned Mary during every meeting {for five weeks/until Christmas/throughout the winter}.

(89) John telephoned Mary during a meeting *{for five weeks/until Christmas/throughout the winter}.

In fact these words can also force eventive-sentences to be universally quantified as shown in (90).

(90) John telephoned Mary {for five weeks/until Christmas/throughout the winter}.

where the event of John's telephoning Mary has a repeated or habitual reading.

In contrast, some other temporal prepositions accept either universal or existential temporal quantification over their modificands such as *in*, *before* and *after* as given in the following examples:

(91) John telephoned Mary every day {in May/before Christmas/after the conference}.

(92) John telephoned Mary one day {in May/before Christmas/after the conference}.

Restricting temporal preposition phrases with some particular quantifications makes systematically generating the semantics of English sentences featuring temporal prepositions possible. Of course, Pratt and Francez's (1997; 2001) theory paid little attention to some important topics such as tense and aspect, as well as the way temporal prepositions employ the time of reference and their role in temporal anaphora.

3.3.2 The Interaction of Prepositions with Tenses and Aspects

In this section, we show that temporal prepositions are not the only means to convey temporal information, but that there are others such as tenses (past, present or future) of the main verb, aspectual classes and grammatical aspects. These also play an essential role in defining the temporal semantics of English sentences. We begin by introducing the interpretations of temporal prepositions depending on their aspectual classes. Then, we discuss the semantics of temporal information concerning the interaction of temporal prepositions with tense and grammatical aspects.

3.3.2.1 Temporal Prepositions and Aspectual Classes

In this section we discuss the effect of the aspectual classes on the interpretation of temporal prepositions. Consider the following sentences:

(93) When John finished his thesis, he went to Australia.

(94) When Mary went to the zoo, she saw some penguins.

Sentence (93) describes an achievement in *when*'s complement and an accomplishment in the main clause. Achievements have the property of being punctual. As a result, we can think of the event in the main clause as occurring immediately after the event in the *when*'s complement. On the other hand, sentence (94) describes an accomplishment in *when*'s complement and an achievement in the main clause. Accomplishments have the property of being durative. As a result, the event in the main clause naturally occurs within the event in *when*'s complement. Thus, the preposition *when* can have more than one temporal relation based on the aspectual classes of its complement and main clause (see Moens and Steedman, 1988; Sandström, 1993; Glasbey, 2004, for more discussion about the interpretation of *when*).

Pratt and Francez (1997) draw attention to the semantics of temporal prepositions involving stative verbs in their main clauses. Consider the following sentences:

(95) Mary slept throughout the lecture.

Sentence (95) describes a state in the main clause. States have the properties of being durative and atelic. Thus, we get

(96) $\imath J(lecture(J) \wedge J \subseteq I, \forall J'(J' \subseteq J \rightarrow (\text{Mary sleep})(J')))$.

which asserts that, the state of Mary's sleeping holds at every time point in the unique interval of the temporal context I occupied by a lecture. Note that stative verbs cannot be combined with prepositions such as the preposition *in* in the following sentence:

(97) * Mary slept in two hours.

Sentence (97) appears odd since the preposition *in* requires a culmination point on its main clause. Thus, the stative verbs fail to cope with the preposition *in*. However, they can come naturally with the preposition *for* as shown in (98).

(98) Mary slept for two hours.

Activities shares a similarity with states – they both have durative and atelic features. Thus, activities can only cope with a temporal preposition that allows its main clauses be universally quantified. For example,

(99) John worked on the paper for 2 hours.

(100) * John wrote the paper for 2 hours.

where sentence (99) describes an activity in the main clause which comes normally with the preposition *for*, since it does not require its main clause to have a telic feature. Thus, sentence (99) asserts that, the activity of John's working on the paper holds at every time point for the past two hours from the time of reference (TOR), as interpreted in (101).

(101) $\forall J(J \subseteq [\text{TOR-2h, TOR}] \rightarrow (\text{John work on the paper})\ (J))$.

On other hand, sentence (100) describes an accomplishment in the main clause which appears unusual with the preposition *for*, because an accomplishment only combines with a temporal preposition that allows its main clause to be existentially quantified.

3.3.2.2 Temporal Prepositions with Tense and Grammatical Aspect

This section shows the effects of tense and the grammatical aspect (perfective or progressive) of main verbs on the semantics of temporal prepositions. Let us begin with the preposition *since*. Pratt and Brée (1993) note that in standard British English the preposition *since* requires its main clause to have a perfective tense. Consider the following sentences:

(102) John has worked on the letter since 9:00.

(103) * John was working on the letter since 9:00.

(104) John has been working on the letter since 9:00.

Sentence (102) describes a present perfective in the main clause. Thus, it asserts that the event of John's working on the letter stretching backward from 9:00 until the TOU as shown in (105). By contrast, sentence (103) uses a past progressive tense in the main clause, thus this sentence appears irregular with the preposition *since*. Instead of (103), we may use a present progressive prefect tense with *since* as shown in (104).

(105) $\forall J(J \subseteq [9{:}00, \text{TOU}] \rightarrow (\text{John work on the letter})(J))$.

The interpretation of the preposition *for* can be affected by tense and grammatical aspect in its main clause. Consider the following sentences:

(106) John will be working on the letter for three hours.

(107) John has worked on the letter for three hours.

Sentence (106) has a future non-perfective in the main clause. Thus, this sentence asserts that the event of John's working on the letter stretches forward for three hours from the TOR as shown in (108). By contrast, sentence (107) has a present perfective in the main clause. Thus, this sentence asserts that, the event of John's working on the letter stretching backward for three hours from the TOR as shown in (109).

(108) $\forall J(J \subseteq [\text{TOR}, \text{TOR+3hr}] \rightarrow (\text{John work on the letter})(J))$.

(109) $\forall J(J \subseteq [\text{TOR-3hr,TOR}] \rightarrow (\text{John work on the letter})(J))$.

Furthermore, even though the temporal preposition *by* has the same meaning as *until*, the temporal preposition *by* can produce the temporal relation *during* if its main clause has a progressive aspect. Consider the following example:

(110) Mary posted the package by 10 o'clock.

(111) Mary was working on her project by 10 o'clock

Sentence (110) has a past tense and a perfective aspect in the main clause. Thus, this sentence asserts that the event of Mary's posting the package occurs before 10 o'clock as shown in (112). By contrast, sentence (111) has a past tense and a progressive aspect in the main clause. Thus, this sentence asserts that Mary was still working on her project at 10 o'clock. Pratt and Francez (1997) state that the complex interactions with aspects leads to some difficulties for giving the proper interpretations for some prepositions.

(112) $\exists J(J \subseteq [\text{TOR}, 10{:}00] \wedge (\text{Mary post the package})(J))$.

Different theories exist in the literature regarding the complex interactions of temporal prepositions with aspects. Moens and Steedman (1988) presented a theory explaining the conversion of one aspectual class to another depending on the verb's grammatical aspect. According to their theory, the perfective converts statives, activities and accomplishments into achievements, placing focus on the consequent state of the described event. Progressives on the other hand convert achievements, accomplishments and statives into activities. Consider the following sentences:

(113) When John arrived at the pub, he drank a bottle of beer.

(114) When John arrived at the pub, he had drunk a bottle of beer.

Sentences (113) and (114) each describe the events of John arriving at the pub and drinking a bottle of beer. Moreover, both events in the given examples have the perfective aspect which describes a situation as a simple whole without an internal structure. However, we have the temporal ordering of these events altered by the presence of the perfect tense in sentence (114). It places focus on the consequent state of the event. Activities and accomplishments are converted to achievements when they occur in the perfective tense. Sentence (113) places the event of John drinking a bottle of beer after his arrival at the pub, but having the main clause event in its perfective form places the focus on the consequent state of the event, we therefore have the event in *when*'s complement placed after the event in the main clause as illustrated in sentence (114).

The progressive converts states, accomplishments and achievements into activities.

(115) ? When John baked the cake, the oven stopped working.

(116) When John was baking the cake, the oven stopped working.

Sentence (115) describes an accomplishment in *when*'s complement, but semantic considerations prevent us from locating the main clause event in the preparatory process of the accomplishment. Sentence (115) is considered odd because if the oven stopped working while John was baking the cake, then it hard to say that John finish baking the cake. When the event in *when*'s complement is however in its progressive form, it is converted to an activity and we therefore have the event in the main clause located within the event in *when*'s complement as illustrated by sentence (116). This allows us therefore to have a temporal inclusion as expected for activities. The results of this study show the temporal relations in temporal prepositions are affected not only by their aspectual classes, but also by tense and grammatical aspects which transform the interpretation of temporal prepositions from a particular relation to another.

In this section, we provide an overview of the semantics of temporal prepositions in English. The purpose of these studies is to determine the temporal semantics concerning the interaction of temporal prepositions with the temporal expressions: tense and aspect. We see how these expressions can change the behaviours of some temporal prepositions. However, for the purpose of this thesis –namely, interpreting protocol specifications in natural languages– we do not need to do justice to every single aspect of the behaviours of temporal prepositions in English.

Chapter 4

Temporal Controlled Natural Language

In this chapter we present *temporal controlled natural language* (TempCNL) and its associated software. In Section 4.1, we show the structure of TempCNL Parsing Engine (TPE) which is used for extracting temporal semantic representations from TempCNL sentences. In Section 4.2, anaphora resolution is presented as a part of the TPE structure to handle the problem of resolving references to earlier or later items in the inputs. In Section 4.3, we describe how we build the TempCNL lexicon for the TPE system. In Section 4.4, an extension of the temporal logic (TPL) is defined as temporal semantic representations for TempCNL sentences. In Section 4.5, our context free grammar (CFG) is constructed for extracting the temporal semantics from TempCNL sentences. In section 4.6, we restrict some of the extended TPL languages in order to make their mapping into LTL and SVA possible (as explained in the next Chapter). At the end of the chapter, we rewrite some of the extended TPL primitives to simplify this translation process.

4.1 The Structure of TempCNL Parsing Engine

A controlled natural language is a formal language with a precisely specified syntax, lexicon and semantics, which, however, resembles a natural language to a sufficient degree that its well-formed sentences are recognizable and (within certain limits) interpretable by a native speaker of that language. Some CNLs are designed for general use, for example ASD (2013) and Avaya Inc. (2004). Other CNLs, by contrast, are designed specifically for translation into formal specification languages, for example Fuchs et al. (1998), Schwitter (2002) and Sowa (2004). Our language, TempCNL, falls into this latter category.

As is common with CNLs, TempCNL departs from natural language in certain, precisely specified, respects, stemming from the fact that it is designed primarily for the specification of communication protocols involving Boolean-valued variables in the context of clocked systems. In such a context, the term "cycle", for example, has a special meaning: one tick of the clock. Likewise, a signal can typically have only two values—variously referred to in practice as "high"/"low", or, equivalently, as "asserted"/"not asserted", or "holding"/"not holding". These terms, though not equivalent in English, in practice have the same meanings in the contexts for which TempCNL is designed; and the semantics of TempCNL reflects this.

Likewise, the primary target-language into which TempCNL is translated, namely, the formal language *SVA* (or, equivalently, *LTL*), has certain syntactic limitations which reflect in obvious ways on the allowed syntax of TempCNL. Thus, for example, *SVA* is fundamentally a propositional language—all quantification is temporal. For instance, we can say that a particular signal, Awid, is low whenever another particular signal, AwidEnable, is low (temporal quantification); however, we cannot say that *all* signals satisfying some property P are low whenever *some* signal satisfying some other property Q is low (atemporal and temporal quantification). Since propositions of the latter kind are not in our target language, they need not be in TempCNL; accordingly, the grammar will not allow them. Thus, the grammar of TempCNL is, in certain respects, greatly simplified in comparison to English. Thus, we can say all TempCNL sentences are acknowledged as English sentences, but that not all English sentences are approved in TempCNL.

Now, let us explain the syntax of TempCNL via examples. Consider the following:

(117) Awvalid occurs when Awid is low;

(118) Every write burst occurs when Awid is low;

(119) The order in which addresses and the data item are produced must match.

Sentence (117) belongs to TempCNL because it can be recognized by our grammar. By contrast, sentence (118) does not belong to TempCNL because the article "Every" is considered as non-temporal quantification. Thus, sentence (118) cannot be translated into *SVA*. Such these examples with non-temporal quantification will be rejected by our grammar rules. Sentence (119) also does not belong to TempCNL because, in the corpora considered in this thesis, very few relative clauses were encountered. Relative clauses are excluded from TempCNL because of the difficulty of distinguishing restrictive from non-restrictive clauses, and because they degrade performance of our parser. They could be considered in future developments.

Given this degree of regimentation of syntax and semantics, the issue arises of course arises as to whether the CNL language we end up with is in fact natural at all—or whether it is not in fact just some formal language like *SVA* with verbose or clumsy syntax, no easier to understand than *SVA*, and twice as difficult to write. The investigations of this Thesis indicate that, for TempCNL, this is not the case. We have striven to design a CNL which really is natural (for the intended users), firstly, by relating it closely to existing theories of temporal semantics in natural language, and secondly—and more importantly—by tailoring it specifically to parse sentences taken from real corpora of specifications.

In this thesis, we have constructed grammar rules for TempCNL using only two corpora that are taken from natural language specifications. These corpora are *Advanced Microcontroller Bus Architecture* in ARM Ltd (2009, 2012a,b) and *Open Core Protocol* in OCP-IP Association (2013). The complete grammar rules are listed in Appendix B and consisted of lexicon, syntax and semantics. It is important to clarify that if a user wants to use the listed grammar, he or she may require to predefine the words or rules related to his or her specific domain.

We extract temporal semantics from TempCNL sentences using the TempCNL Parsing Engine (TPE). The TPE system is written in SWI-Prolog and based on the main components in Figure 4.1.1. The main components are anaphora resolution and semantic parsing. The semantic parsing depends on two components which are the lexicon and the context-free grammar (CFG).

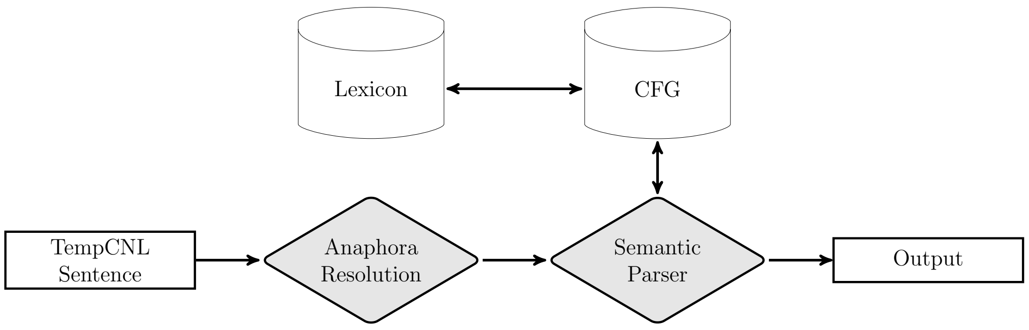


Figure 4.1.1: The TPE structure.

Based on the TPE structure in Figure 4.1.1, any TempCNL sentence will be tested by an anaphora resolution to check if any anaphora occurs. Then, we replace the anaphor with its antecedent. Finally, we parse the input without any anaphora using our semantic parser to extract the temporal semantics.

4.2 Anaphora Resolution

This section presents our method for resolving references to earlier or later items in the text using anaphora resolution. We choose one of the most efficient anaphora resolution systems which is JavaRAP Qiu et al. (2004). JavaRAP resolves third person pronouns, lexical anaphors, and recognises expletive pronouns. Note that expletive (or dummy) pronouns are words that refer to nothing particular, instead allowing the sentences to function correctly in a grammatical context. Based on JavaRAP's evaluation in Qiu et al. (2004), JavaRAP has been tested on the MUC-6 dataset (see Grishman and Sundheim, 1996) which contains 235 lexical anaphors and third person pronouns.

The result was that the JavaRAP labelled 136 correctly and the accuracy was 57.9%. The low accuracy of JavaRAP is mainly due to the limitation of the performance of Charniak's (2000) parser. JavaRAP uses Charniak's (2000) parser for taking text with sentence delimitation as input and generating a parse tree. In the TPE system, our aim is to parse sentences specifying system requirements. These types of sentences are usually written shorter than sentences in the MUC-6 dataset. Therefore, the performance of JavaRAP in term of accuracy is higher than that reported in Qiu et al. (2004). Section 6.2.2 shows JavaRAP's evaluation on a corpus that contains English sentences explaining system requirements.

Let us show how we use the JavaRAP tool to resolve references. We first parse the inputs through the JavaRAP tool to produce a list of anaphora-antecedent pairs as outputs. Then we replace all anaphors with their antecedents in the given inputs. For example,

(120) When **Acvalid** is asserted, **it** must remain asserted until Acready is asserted.

(121) If **they** are active, **MReset and SReset** must stay active at least 16 consecutive cycles.

In sentence (120), the third-person singular pronoun "it" refers to its antecedent "Acvalid". In contrast, in sentence (121), the third-person plural personal pronoun "they" refers to its postcedent "MReset and SReset". Using the JavaRAP tool allows us to generate a list of anaphora-antecedent pairs as shown in Table 4.2.1.

Antecedent or Postcedent		Anaphora
Acvalid	⇐	it
MReset and SReset	⇐	they

Table 4.2.1: A list of anaphora-antecedent pairs.

We take the output shown in Table 4.2.1 and replace any anaphora with its antecedent or its postcedent in the original sentences such as sentences (120) and (121) and reproduce sentences without any anaphora such as the following sentences:

(122) When **Acvalid** is asserted, **Acvalid** must remain asserted until Acready is asserted.

(123) If **MReset and SReset** are active, **MReset and SReset** must stay active at least 16 consecutive cycles.

Note that the above method is not applicable if the antecedent preforms as a quantified noun phrase such as (124) since if we replace “he” with “a man” such as in (125), it does not mean the same as (124).

(124) A man came into the room. He sat down.

(125) A man came into the room. A man sat down.

Luckily, there are no examples of pronouns referring back to indefinite noun phrases in the corpora that we work on. More importantly, the TPE grammar will not allow non-temporal quantification such as “A” in our TempCNL as explained in Section 4.1.

No anaphora resolution method is perfect, of course. However, in practice, the JavaRAP tool works well. For the data sets we considered, all sentences are short and have particularly simple anaphoric structure with little ambiguity.

Of course, the exact manner in which the JavaRAP tool works depends on the details of the algorithm involved, and is not specified as part of our definition of TPE. This, of course, represents a weakness in our system, one which should be repaired by deciding on the best anaphora-resolution strategy from the point of view of the design of a controlled natural language, and then implementing that strategy. We plan to address this issue in future work. We note however, that the JavaRAP tool works well on the corpora we have considered.

4.3 TempCNL lexicon

This section describes how we build the TempCNL lexicon for the TPE system. Our lexicon contains detailed information about the words belonging to the possible syntactic categories. We define the lexical entry using one of the following formats:

```
lexEntry(Cat,[symbol:Sym,syntax:Word,num:Num]).
lexEntry(Cat,[syntax:Word,type:Q]).
```

where `Cat` is the syntactic category, `Word` the strings of the word, `Sym` the relation symbol or the constant of the given word, and `Num` lists the singular or plural information for nouns and verbs. In the second lexical entry, we have the variable `Q` which lists additional information about the quantification type for a determiner or the type of coordination, as discussed below.

Now let us show the TempCNL lexical rules. The lexical rules include only terminal categories on the right-hand side, and they are classified into two classes: closed class (or predefined function words) and open class (or content words). Let us discuss each class in detail.

4.3.1 Closed class lexicon

Closed class lexicon consists of 6 different categories: determiner, auxiliary verb, negation, coordination, temporal adjective and temporal preposition.

1. **Determiners** are defined in the following form:

```
det([type:Q])-->
   {lexEntry(det,[syntax:Word,type:Q])}, Word.
```

where the variable `Q` uses to store the quantification type that determiner has in its lexicon entry as shown, below:

```
lexEntry(det,[syntax:[every],type:forall]).
lexEntry(det,[syntax:[a],type:exists]).
lexEntry(det,[syntax:[the],type:def]).
```

The tags `forall`, `exists` and `def` denote universal, existential and definite quantifications, respectively. These tags are used to give some syntactic restrictions to certain phrase categories as we shall see later in the following section. In the TempCNL lexicon, there are six determiners as shown in Table 4.3.1 with their associated quantification types.

Determiner	Quantification type
any, every or all	forall
a, an or some	exists
the	def

Table 4.3.1: A list of determiners with their quantification types.

2. **Auxiliary verbs** are defined in the following form:

```
aux([num:Num])-->
   {lexEntry(aux,[syntax:Word,num:Num])},Word.
```

The TempCNL lexicon has 19 auxiliary verbs as follows:

are	is	was	were	be	being	been	have	has	had
may	must	might	can	could	would	should	shall	will	

Here are some of their lexical entries as follows:

```
lexEntry(aux,[syntax:[is],num:sg]).
lexEntry(aux,[syntax:[are],num:pl]).
lexEntry(aux,[syntax:[must],num:_]).
```

The `sg` denotes that the auxiliary verb "is" is singular while the `pl` denotes that the auxiliary verb "are" is plural. Note that the modal auxiliary "must" does not have third-person singular present tense form. Thus, we leave this field empty.

Note that, in our TempCNL grammar, we allow multiple auxiliaries such as the following sentences:

Subject	MODAL	PERF	PROG	PASS	LEXICAL
The game				was	played.
The game			was	being	played.
The game			has	been	played.
The game		has	been	being	played.
The game	might	have	been	being	played.

Thus, we add another lexicon rule to handle multiple auxiliaries as follows:

```
extraAux  -->  {lexEntry(extraAux,[syntax:Word])},Word.
```

This category has 6 lexical entries as follows:

```
lexEntry(extraAux,[syntax:[be]]).
lexEntry(extraAux,[syntax:[been]]).
lexEntry(extraAux,[syntax:[being]]).
lexEntry(extraAux,[syntax:[have, been]]).
lexEntry(extraAux,[syntax:[have, being]]).
lexEntry(extraAux,[syntax:[been, being]]).
```

Those lexical entries defined based on the verb forms for the possible combinations of verb phrases with multiple auxiliaries in Table 4.3.2.

modal	+	perfect have	+	passive be	+	Verb
modal	+	perfect have	+	progressive be	+	Verb
perfect have	+	progressive be	+	passive be	+	Verb
modal	+	progressive be	+	passive be	+	Verb

Table 4.3.2: The possible combinations of verb phrases with multiple auxiliaries.

3. **Negations** are defined in the following form:

   ```
   neg --> {lexEntry(neg,[syntax:Word])},Word.
   ```

   There are only three lexical entries for negations and they are as follows:

   ```
   lexEntry(neg,[syntax:[no]]).
   lexEntry(neg,[syntax:[not]]).
   lexEntry(neg,[syntax:[never]]).
   ```

4. **Coordination** is classified into three different types of lexical categories since we allow coordination for sentences, noun phrases and temporal nominal phrases. We define their closed-class lexical rules as follows:

   ```
   ipcoord --> {lexEntry(coord,[syntax:Word,type:Type])}, Word.
   npcoord --> {lexEntry(coord,[syntax:Word,type:Type])},Word.
   tncoord --> {lexEntry(coord,[syntax:Word,type:Type])},Word.
   ```

   The `ipcoord` category denotes the coordination for sentences, the `npcoord` category denotes the coordination for noun phrases, and the `tncoord` category denotes the coordination for temporal nominal phrases. Note that all the above rules have the same syntactic category `coord`; however each rule will have different semantic interpretations as we shall see later in Section 4.5.5. There are only two lexical entries for coordination as shown, below:

   ```
   lexEntry(coord,[syntax:[and],type:conj]).
   lexEntry(coord,[syntax:[or],type:disj]).
   ```

5. **Temporal adjectives** are defined in the following form:

   ```
   tAdj --> {lexEntry(tadj,[symbol:Sym,syntax:Word])}, Word.
   ```

   There are only two lexical entries for the order-denoting adjectives. These entries are as follows:

   ```
   lexEntry(tadj,[symbol:f,syntax:[first]]).
   lexEntry(tadj,[symbol:l,syntax:[last]]).
   ```

6. **Temporal Prepositions** are classified into two different types of lexical categories: temporal prepositions with nominal complements and temporal prepositions with clausal complements.

   (a) Closed-class lexical rule for temporal prepositions with nominal complements is defined as follows:

```
tpn([qclause:Q,mclause:M])-->
   {lexEntry(tpn,[symbol:Sym,syntax:Word,
   qclause:Q,mclause:M])},Word.
```

   The TempCNL lexicon has 13 temporal prepositions that can select nominal complements, and these temporal prepositions are as follows:

within	at	during	on	for	in	throughout
until	by	since	before	after	until after	

   (b) Closed-class lexical rule for temporal prepositions with clausal complements is defined as follows:

```
tps([qclause:Q,mclause:M])-->
   {lexEntry(tps,[symbol:Sym,syntax:Word,
   qclause:Q,mclause:M])},Word.
```

   The TempCNL lexicon has 11 temporal prepositions that can select clausal complements, and these temporal prepositions are as follows:

whilst	when	while	whenever	once	until
before	by the time	after	since	until after	

   Note that there are some prepositions that can be either temporal or non-temporal. However, there are very low probabilities for ambiguity occurring, since temporal prepositions will only combine with temporal nouns in our grammar as we shall see in Section 4.5.

Note that some temporal prepositions enforce their complements and main clauses to be restricted with particular quantifications. Thus, we add two variables `Q` and `M` in the `tpn` and `tps` categories to restrict the complements and main clauses with the quantification types that associated with those temporal prepositions. These restrictions provide adequate systematic treatments of which sentences are acceptable in our TempCNL grammar based on the common use of the temporal preposition in English. For example, the temporal prepositions *until* and *by* require their complements to be definitely quantified as shown in (83) and (84), which we repeat here for convenience as follows, respectively:

(126) John telephoned Mary every day until {the exam/* every exam/* an exam}.

(127) John telephoned Mary by {the exam/* every exam/* an exam}.

Moreover, the temporal preposition *until* requires its main clause to be universally quantified while the temporal preposition *by* requires its main clause to be existentially quantified. Thus, we write their lexical entries as follows:

```
lexEntry(tpn,[symbol:before,syntax:[by],
          qclause:def,mclause:exists]).
lexEntry(tpn,[symbol:before,syntax:[until],
          qclause:def,mclause:forall]).
```

Both lexicon entries have, in the `symbol` filed, the temporal order relation `before`. This is due to the initial meaning of the temporal prepositions *by* and *until* which locates one event or state before another in time. The temporal relation `before` will be used later to produce the meanings of those temporal prepositions. The rest of lexical rules for temporal prepositions including their restrictions is listed in Appendix B.

4.3.2 Open class lexicon

Open class lexicon consists of 9 different categories: temporal nouns, proper names, cardinal numbers, adverbs, adjectives, inflections, intransitive verbs, transitive verbs and ditransitive verbs.

1. **Temporal Nouns** are defined in the following form:

   ```
   tn([num:Num])-->
      {lexEntry(tnoun,[symbol:Sym,num:Num,syntax:Word])},Word.
   ```

   We define the lexical entry for a temporal noun such as "cycles" and "transaction" as follows:

   ```
   lexEntry(tnoun,[symbol:cycle,num:pl,syntax:[cycles]]).
   lexEntry(tnoun,[symbol:transaction,num:sg,syntax:[transaction]]).
   ```

2. **Proper names** are classified into two different versions: one for those that stand by itself (e.g. "John", "Microsoft") and one for those that require a preceeding definite article (e.g. "the Atlantic", "the USA"). We define them as follows, respectively:

   ```
   pn([num:Num])-->
      {lexEntry(pn,[symbol:Sym,num:Num,syntax:Word])}, Word.
   pn_def([num:Num])-->
      {lexEntry(pn_def,[symbol:Sym,num:Num,syntax:Word])}, Word.
   ```

   We write the lexical entry for proper names such as "John" and "the USA" as follows:

   ```
   lexEntry(pn,[symbol:‘John’,num:sg,syntax:[‘John’]]).
   lexEntry(pn_def,[symbol:‘USA’,num:sg,syntax:[‘USA’]]).
   ```

   Note that some proper names can also be declared in both of those categories (e.g. "(the) Awvalid", "(the) Eiffel Tower").

3. **Cardinal numbers** are defined in the following form:

```
num   --> {lexEntry(number,[symbol:Sym,syntax:Word])}, Word.
```

We write the lexical entry for numbers such as "4" and "six" (which is written in words) as follows:

```
lexEntry(number,[symbol:'4',syntax:['4']]).
lexEntry(number,[symbol:'6',syntax:[six]]).
```

4. **Adverbs** are defined in the following form:

```
adv --> {lexEntry(adv,[syntax:Word])}, Word.
```

We write the lexical entry for adverbs such as *quickly* and *exactly* as follows:

```
lexEntry(adv,[syntax:[exactly]]).
lexEntry(adv,[syntax:[eventually]]).
```

5. **Adjectives** are defined in the following form:

```
adj --> {lexEntry(adj,[symbol:Sym,syntax:Word])},Word.
```

We write the lexical entry for adjectives such as *high* and *low* as follows:

```
lexEntry(adj,[symbol:high,syntax:[high]]).
lexEntry(adj,[symbol:low,syntax:[low]]).
```

6. **Inflections** refer here to the extra letter or letters added to verbs. Inflectional endings can indicate the tense of verbs. Inflections are defined in the following forms:

```
i([symbol:Root,num:sg]) -->
  {morph_type(Word, Root,-s), lexEntry(iv,[syntax:Root])}, [Word].
i([symbol:Root,num:_]) -->
  {morph_type(Word, Root,none), lexEntry(iv,[syntax:Root])},[Word].
i([symbol:Root,num:_]) -->
  {morph_type(Word, Root,-ed), lexEntry(iv,[syntax:Root])}, [Word].
```

Note that, in the above rules, the lexical entry belongs to the intransitive verb (`iv`) category. However, we can apply the above rules to the categories transitive and ditransitive verbs in the similar way. The main purpose of the inflection category is to extract the root of the verb and move it to the appropriate position where we can extract its meaning. For more clarification, see the phrase structure of sentence (149) in Figure 4.5.2). Note that, in the `symbol` field, the variable `Root` stores the base form of the given verb.

To extract inflectional suffixes of the verb, we use the function `morph_type`. This function is built in a morphological analysis tool called ProNTo (developed in Schlachter (2003)). The ProNTo tool is written in Prolog which make it easy for us to integrate the ProNTo tool with the TPE system. Figure 4.3.1 illustrates how to extract inflectional suffixes of verbs "played" and "remains" using the function `morph_type`.

```
1. morph_type(played, R, S)
2. Output:
  Root (R) = play
  Suffix (S) = -ed
```

```
1. morph_type(remains, R, S)
2. Output:
  Root (R) = remain
  Suffix (S) = -s
```

Figure 4.3.1: Extracting inflectional suffixes from verbs.

After that we check if the verbs "played" and "remains" have lexical entries in our lexicon as shown below:

```
lexEntry(iv,[syntax:[remain]]).
lexEntry(tv,[syntax:[play]]).
```

7. **Intransitive verbs** are classified into two categories: verbs that do not take direct objects and verbs that are followed by adjectives. We define both lexicon categories as shown in the following forms, respectively:

```
iv([symbol:Word,mclause:M]) -->
   {morph_type(Word,Root,Suffix), grammatical_aspect(Suffix,GrAsp),
   aspectual_class(Root,Type), quantifier_type(GrAsp,Type,M),
   lexEntry(iv,[syntax:Root])},[].
```

```
iv([symbol:[],mclause:M]) -->
   {morph_type(Word,Root,Suffix), grammatical_aspect(Suffix,GrAsp),
   aspectual_class(Root,Type), quantifier_type(GrAsp,Type,M),
   lexEntry(iv,[syntax:Root])}, [Word].
iv_adj([symbol:Word,mclause:M]) -->
   {morph_type(Word,Root,Suffix), grammatical_aspect(Suffix,GrAsp),
   aspectual_class(Root,Type), quantifier_type(GrAsp,Type,M),
   lexEntry(iv,[syntax:Root])},[].
iv_adj([symbol:[],mclause:M]) -->
   {morph_type(Word,Root,Suffix), grammatical_aspect(Suffix,GrAsp),
   aspectual_class(Root,Type), quantifier_type(GrAsp,Type,M),
   lexEntry(iv,[syntax:Root])}, [Word].
```

All the above rules are defined in the same way, except the first and third rules which do not take the next word in the given sentences since the variable `Word` in the `symbol` feature has the root of the verb that is passed from the inflection category. Note that all the above rules use the same part-of-speech tag which is `iv`; however the rules for the `iv` category will have different semantic interpretations than the rules for the `iv_adj` category as we shall see later in Section 4.5.2.2. The variable `M` stores the quantification type of a verb.

To assign the proper quantification types for verbs, we must have additional rules to extract two important features: the grammatical aspects (simple, perfective, progressive) and the aspectual classes (state, event). The grammatical aspects of verbs tell us whether the event verb has a culmination point or not. In contrast, the aspectual classes tell us whether events extend in time or occur at an instantaneous time (more details are given in Section 3.2). We use those features to select a quantifier type (existential or universal) for a verb. For example, consider the following sentences:

(128) Awvalid is asserted

(129) Awvalid remains high.

In (128), the verb "asserted" is an event verb, and it has grammatically passive construction; however in this case we are going to treat it like a perfective aspect because that what it makes most sense in natural language protocol specifications. In (129), the verb "remains" is a state verb and has a simple aspect. We assign existential quantification to the verb "asserted" since any event verb with the perfective aspect indicates that the event has a culmination point which occurs instantaneously; and universal quantification to the verb "remains" since any state verb with the simple aspect indicates that the event has no culmination point and extends in time. Table 4.3.3 shows our quantifier selection decision for verbs based on the grammatical aspects and the aspectual classes.

Grammatical Aspects	×	Aspectual classes	→	Quantified types
Simple aspect	×	State class	→	Universal type
Simple aspect	×	Event class	→	Existential type
Perfective aspect	×	State class	→	Universal type
Perfective aspect	×	Event class	→	Existential type
Progressive aspect	×	State class	→	Universal type
Progressive aspect	×	Event class	→	Universal type

Table 4.3.3: Our method to select a quantified type for a verb.

Let us discuss how to extract the grammatical aspects of verbs. Then we show how we set the lexicon rules for aspectual classes.

We take advantage of the ProNTo tool to extract the inflectional suffixes as shown in the given examples in Figure 4.3.1. Then, we can obtain those aspects of verbs using the simple Prolog rule "`grammatical_aspect(Suffix, GrAsp)`" that retrieves the associated aspects of the given inflectional suffixes. Table 4.3.4 shows a summary of the inflectional suffixes that we require to determine the grammatical aspects.

Suffix	Grammatical change	Grammatical Aspects
none	none	simple Aspect
-ed	past tense/past participle	perfect Aspect
-en	past participle (irregular)	perfect Aspect
-ing	continuous/progressive	progressive Aspect

Table 4.3.4: The inflectional suffixes with their associated aspects.

As shown in Table 4.3.4, every inflectional suffix is associated with a particular aspect. The "none" word means that there is no suffix on the given verbs.

Now let us show how we build the rules for the aspectual classes of verbs. As mentioned in Section 3.2, in this thesis, the only thing we really require is to make a distinction between states and events for our problem since our purpose is to handle natural language protocol specifications. Therefore, we have two class names: state and event. For those classes, we write the rules, for example, to the verbs in sentences (128) and (129) as shown, below:

```
aspectual_class(assert,event).
aspectual_class(remain,state).
```

Finally, when we extract the grammatical aspect (its variable `GrAsp`) and the aspectual class (its variable `Type`) for the given verb, we select the suitable quantifier for that verb using the Prolog function "`quantifier_type(GrAsp,Type, M)`" to retrieve the quantifier type (its variable `M`) for the verbs based on our quantifier selection decision in Table 4.3.3. Here are the full details of the verbs "asserted" and "remain" based on our set-up lexical rules for the `iv` and `iv_adj` categories:

```
iv([symbol:[],mclause:exists]) -->
 {morph_type(asserted,assert,-ed),grammatical_aspect(-ed,perf),
 aspectual_class(assert,event),quantifier_type(perf,event,exists),
 lexEntry(iv,[syntax:assert])}, [asserted].
iv_adj([symbol:[],mclause:forall]) -->
 {morph_type(remain,remain,none),grammatical_aspect(remain,state),
 aspectual_class(remain,state),quantifier_type(simple,state,forall)
 ,lexEntry(iv,[syntax:remain])}, [remain].
```

As noted above, the lexical rules for the `iv` and `iv_adj` categories are complex. Thus, form now on, we write the following forms instead of the above forms for simplicity's sake when we discuss the lexical rules for these two categories.

```
iv([symbol:[],mclause:exists]) -->  [asserted].
iv_adj([symbol:[],mclause:forall]) --> [remain].
```

8. **Transitive verbs** are verbs that can take direct objects. We define its lexical rules as follows:

```
tv([symbol:Word,mclause:M]) -->
   {morph_type(Word,Root,Suffix), grammatical_aspect(Suffix,GrAsp),
   aspectual_class(Root,Type), quantifier_type(GrAsp,Type,M),
   lexEntry(tv,[syntax:Root])}, [].
tv([symbol:[],mclause:M]) -->
   {morph_type(Word,Root,Suffix), grammatical_aspect(Suffix,GrAsp),
   aspectual_class(Root,Type), quantifier_type(GrAsp,Type,M),
   lexEntry(tv,[syntax:Root])}, [Word].
```

   Here is an example of a transitive verb and its lexical rules:

```
tv([symbol:[],mclause:forall]) -->
   {morph_type(eating, eat, -ing), grammatical_aspect(-ing, prog),
   aspectual_class(eat,event), quantifier_type(prog,event,forall),
   lexEntry(tv,[syntax:eat])}, [eating].
```

9. **Ditransitive verbs** are verbs which take a subject and two objects. We define its lexical rules as follows:

```
dtv([symbol:Word,mclause:M]) -->
   {morph_type(Word,Root,Suffix), grammatical_aspect(Suffix,GrAsp),
   aspectual_class(Root,Type), quantifier_type(GrAsp,Type,M),
   lexEntry(dtv,[syntax:Root])},[].
dtv([symbol:[],mclause:M]) -->
   {morph_type(Word,Root,Suffix), grammatical_aspect(Suffix,GrAsp),
   aspectual_class(Root,Type), quantifier_type(GrAsp,Type,M),
   lexEntry(dtv,[syntax:Root])}, [Word].
```

   Here is an example of a ditransitive verb and its lexical rules:

```
dtv([symbol:[],mclause:exists]) -->
 {morph_type(give, give, none),grammatical_aspect(none,simple),
 aspectual_class(give,event),quantifier_type(simple,event,exists),
 lexEntry(dtv,[syntax:give])}, [give].
```

Note that the lexical rules for transitive and ditransitive verbs are similar to the rules of intransitive verbs except its lexical entry. Again, for simplicity's sake, we write the lexical rules for transitive and ditransitive verbs in the following forms when we discuss them later.

```
tv([symbol:[],mclause:forall]) -->  [eating].
dtv([symbol:[], mclause:exists]) --> [give].
```

The TempCNL lexicon consists of 814 domain-specific words that taken from *Advanced Microcontroller Bus Architecture* in ARM Ltd (2009, 2012a,b) and *Open Core Protocol* in OCP-IP Association (2013).

4.4 TPL^+ Semantics

In this section we present the syntax and semantics of the logic TPL^+, which is an extension of TPL as defined in Pratt-Hartmann (2005). The TPE system translates TempCNL sentences unambiguously into TPL^+.

Let us begin with the syntax of TPL^+ as follows.

In the sequel, let E be a finite set. We refer to the elements of E as *event-atoms.*

Definition 4.4.1. Let e range over the set E of event-atoms. We define the categories of *event-relation* α, TPL^+-formula ψ as follows:

$\alpha := e \mid e^f \mid e^l;$

$\theta := \top \mid \langle e\rangle_= \theta \mid \; |e\rangle_= \theta \mid \langle e^{\leftrightharpoons}\rangle_= \theta \mid [\![e]\!];$

$\psi := \theta \mid \neg\psi \mid \langle e\rangle_= \psi \mid [e]_= \psi \mid \{\alpha\}_= \psi \mid \; |n\rangle_* \psi \mid [e]_< \psi \mid [e]_> \psi \mid \{\alpha\}_< \psi \mid \{\alpha\}_> \psi \mid \{\alpha\}_{<+} \psi \mid$
$\quad \psi \wedge \psi' \mid \psi \vee \psi'.$

The symbols $<$, $>$, $=$ denote the temporal order relations *before*, *after* and *during*, respectively. Moreover, the bracket-pairs $\langle\ \rangle$, $[]$, $\{\ \}$, $|\ \rangle$ denote existential , universal, definite and initial existential quantifiers, respectively, and the symbol ($\overleftarrow{\ }$) indicates that its argument holds at the time of evaluation if and only if it held at the previous time point. The symbol $(*)$ denotes the repetition of ψ a given number of times. The Boolean connectives $\rightarrow$, $\leftrightarrow$, *and* $\perp$ are understood in the usual way. Finally, we may remove the $_=$ subscript (which denotes the *during* relation) and consider any formula omitting its binary relation as a formula with the *during* relation. For example, we write $\langle e\rangle\top$ and $\{\alpha\}\psi$ rather than $\langle e\rangle_=\top$ and $\{\alpha\}_=\psi$, respectively.

Recall $\mathcal{I}_\mathcal{R}$ from Definition 2.6.1 is a set of intervals, where an interval is a closed, bounded, non-empty subset of the real numbers. Analogously, we define $\mathcal{I}$ in the following definition.

Definition 4.4.2. We take an interval to be a closed, bounded and non-empty subset of the natural numbers. More formally we say that an interval is a pair $[a, b]$ such that $a, b \in \mathbb{N}$ and $a \leq b$. We denote the set of all intervals $\{[a, b] : a \leq b \wedge a, b \in \mathbb{N}\}$ by $\mathcal{I}$, and we use the letters I, J, ... with or without decorations as variables ranging over $\mathcal{I}$.

From now on, we are going to consider intervals not over the real numbers but over the natural numbers. Thus, the logic TPL^+ is a slightly different from TPL since TPL^+ only makes sense over discrete time.

Here, we also define the functions *start* and *end* return the beginning and end points of an interval, respectively. For example, $start([a, b])$ returns a and $end([a, b])$ returns b. Moreover, let us recall the terms init(J,I) and fin(J,I) which denote the initial segment of I up to the beginning of J, and the final segment of I from the end of J, respectively.

Before giving the formal semantics of TPL^+ formulas, we will define the notion of TPL^+-interpretation.

Definition 4.4.3. A TPL^+-interpretation $\mathfrak{A}$ (henceforth, interpretation) is a finite subset of $\mathcal{I} \times E$. For any $J \in \mathcal{I}$, we write $\mathfrak{A}(J)$ for $\{e \in E \mid \langle J, e\rangle \in \mathfrak{A}\}$, and for any $e \in E$, we write $\mathfrak{A}(e)$ for $\{J \in \mathcal{I} \mid \langle J, e\rangle \in \mathfrak{A}\}$.

Intuitively, $\langle J, e\rangle \in \mathfrak{A}$ means that an event of type e occurred over the interval J.

Let us now discuss the interpretation of event-relations. We recall the following terminology from Pratt-Hartmann (2005) which defines the meanings of the words *first* and *last* to apply to event-types of which there is no unambiguously *first* or *last* instance.

Definition 4.4.4. Let α be an *event-relation*, $\mathfrak{A}$ an interpretation, and $I, J \in \mathcal{I}$. We define $\mathfrak{A} \models_{I,J} \alpha$ by cases as follows:

1. $\mathfrak{A} \models_{I,J} e$ iff $J \subset I$ and $e \in \mathfrak{A}(J)$;
2. $\mathfrak{A} \models_{I,J} e^f$ iff $\mathfrak{A} \models_{I,J} e$ and there is no J' such that $\mathfrak{A} \models_{I,J'} e$ and start$(J') <$ start(J);
3. $\mathfrak{A} \models_{I,J} e^l$ iff $\mathfrak{A} \models_{I,J} e$ and there is no J' such that $\mathfrak{A} \models_{I,J'} e$ and end$(J) <$ end(J');

It is evident that, since $\mathfrak{A}$ is finite, if there exists any $J \subset I$ such that $\langle J, e\rangle \in \mathfrak{A}$, then the *first* and *last* intervals such J exist and are unique.

Let us now turn to the definitions of the truth-conditions for formulas in $\mathcal{TPL}^+$.

Definition 4.4.5. Let ϕ be a formula, $\mathfrak{A}$ an interpretation, and $I \in \mathcal{I}$. We define $\mathfrak{A} \models_I \phi$ recursively as follows:

1. $\mathfrak{A} \models_I \langle e\rangle\top$ iff for some $J \subseteq I, J \in \mathfrak{A}(e)$;
2. $\mathfrak{A} \models_I |e\rangle\top$ iff for some $J \subseteq I, J \in \mathfrak{A}(e)$ and J is an initial interval of I;
3. $\mathfrak{A} \models_I \langle e^{\Leftarrow}\rangle\top$ iff $\mathfrak{A} \models_{[start(I),start(I)]} e \Leftrightarrow \mathfrak{A} \models_{[start(I)\text{-}1,start(I)\text{-}1]} e$;
4. $\mathfrak{A} \models_I [\![e]\!]$ iff for all $J \subseteq I$, $J \in \mathfrak{A}(e)$;
5. $\mathfrak{A} \models_I \langle e\rangle\psi$ iff for some $J \subseteq I, J \in \mathfrak{A}(e)$ and $\mathfrak{A} \models_J \psi$;
6. $\mathfrak{A} \models_I [e]\psi$ iff for all $J \subseteq I, J \in \mathfrak{A}(e)$ implies $\mathfrak{A} \models_J \psi$;

7. $\mathfrak{A} \models_I \{\alpha\}\psi$ iff there is a unique $J \subset I$ such that $J \in \mathfrak{A}(\alpha)$, and for that $J, \mathfrak{A} \models_J \psi$;

8. $\mathfrak{A} \models_I |n\rangle_* \psi$ iff $\mathfrak{A} \models_{[start(I), start(I)+n]} \psi$ and $|I| \geq n$, where n is a number;

9. $\mathfrak{A} \models_I [e]_> \psi$ iff for all $J \subset I$, $J \in \mathfrak{A}(e)$ implies $\mathfrak{A} \models_{[end(J)+1, end(J)+1]} \psi$;

10. $\mathfrak{A} \models_I [e]_< \psi$ iff for all $J \subset I$, $J \in \mathfrak{A}(e)$ implies $\mathfrak{A} \models_{[start(J)-1, start(J)-1]} \psi$;

11. $\mathfrak{A} \models_I \{\alpha\}_< \psi$ iff there is a unique $J \subset I$ such that $J \in \mathfrak{A}(\alpha)$, and for that $J, \mathfrak{A} \models_{init(J,I)} \psi$;

12. $\mathfrak{A} \models_I \{\alpha\}_> \psi$ iff there is a unique $J \subset I$ such that $J \in \mathfrak{A}(\alpha)$, and for that $J, \mathfrak{A} \models_{fin(J,I)} \psi$;

13. $\mathfrak{A} \models_I \{\alpha\}_{<+} \psi$ iff there is a unique $J \subset I$ such that $J \in \mathfrak{A}(\alpha)$, and for that $J, \mathfrak{A} \models_{[start(I), end(J)+1]} \psi$;

14. $\mathfrak{A} \models_I \neg\psi$ *iff* it is not true that $\mathfrak{A} \models_I \psi$.

15. $\mathfrak{A} \models_I \psi \wedge \psi'$ *iff* $\mathfrak{A} \models_I \psi$ and $\mathfrak{A} \models_I \psi'$.

16. $\mathfrak{A} \models_I \psi \vee \psi'$ *iff* $\mathfrak{A} \models_I \psi$ or $\mathfrak{A} \models_I \psi'$.

In Definition 4.4.5, the TPL^+ formulas $|e\rangle\top$, $\langle e^{\leftleftarrows}\rangle\top$, $[\![e]\!]$, $|n\rangle_*\psi$, $[e]_>\psi$, $[e]_<\psi$ and $\{\alpha\}_{<+}\psi$ are defined in this thesis while the rest of them have already been introduced in Pratt-Hartmann (2005). The motivation behind these extensions is to (i) handle more temporal constructions occur commonly in English (such as *throughout*, *until after*, *since*, *stable*, *etc*), and (ii) provide a suitable temporal logic that has sufficient expressive power for hardware specifications.

Note that, in Definition 4.4.5, there is a difference between double square bracket $[\![]\!]$ in 4 and the square bracket [ ] in 6. The $[\![e]\!]$ entails that all subintervals of the interval of evaluation must satisfy e whereas the $[e]\psi$ entails that all intervals which satisfy e make ψ true.

Let us give some examples of how TPL^+ interprets English sentences having temporal expressions and relate these interpretations to those given using ITL (see the formal semantics of ITL in Section 2.6). Consider the following sentences:

(130) Awid is asserted

(131) Awid is asserted during every cycle

(132) Awid is asserted during every cycle until Awvalid goes high.

The semantics of the above sentences in TPL^+ are:

(133) $\langle assert(Awid)\rangle\top$

(134) $[cycle]\langle assert(Awid)\rangle\top$

(135) $\{high(Awvalid)\}_<[cycle]\langle assert(Awid)\rangle\top$.

From Definition 4.4.5, we see that the truth-conditions of the above TPL^+-formulas correspond exactly to ITL formulas as shown below, respectively

(136) $\exists J_0(assert(Awid, J_0) \wedge J_0 \subseteq I)$

(137) $\forall J_1(cycle(J_1) \wedge J_1 \subseteq I \to \exists J_0(assert(Awid, J_0) \wedge J_0 \subseteq J_1))$

(138) $\imath J_2(high(Awvalid, J_2) \wedge J_2 \subseteq I,$
$\forall J_1(cycle(J_1) \wedge J_1 \subseteq init(J_2, I) \to \exists J_0(assert(Awid, J_0) \wedge J_0 \subseteq J_1)))$.

We derive the ITL formula (136) from the TPL^+-formula (133) using the clause (1), the ITL formula (137) from the TPL^+-formula (134) using the clauses (6) and (1), and the ITL formula (138) from the TPL^+-formula (135) using the clauses (11), (6) and (1). Note that the interval variables J_0, J_1 and J_2 do not appear in TPL^+ formulas because the semantics of the operators $\{\}, [], \langle\rangle$ *etc* in Definition 4.4.4 bind those interval variables over which the formulas in their scopes are evaluated. On the other hand, in

ITL formulas these variables become arguments of the predicates such as *assert*, *cycle* and *high* as shown in formulas (136)–(138). ITL has an extra argument place that is filled by interval variables. It is important to mention that we do not intend to translate TPL^+ formulas from ITL formulas since (1) they are equivalent, (2) TPL^+ is easier to extract from English and translate it into LTL and SVA as we shall see in Chapter 5.

The meanings of formulas (136)–(138) were explained previously in (29), (30), and (34), respectively. Note that, in this section, ITL formulas describe over discrete time rather than continuous time.

Now, let us provide interpretations for some temporal expressions that often occur in natural language specifications. Consider the following sentences:

(139) Awvalid remains high for three cycles

(140) After the last ACK, Awid must eventually occur.

(141) Awvalid must be stable once Acvalid goes high

Before we discuss the meaning representations of sentences (140) – (141), we want to point out that the words *stable* and *cycles with cardinal number* have special meanings in hardware languages such as SVA. Thus, the adjective *stable* is assigned to $\langle e^{\Leftarrow}\rangle\top$ which is defined in clause (3) of Definition 4.4.5.

Moreover, if the word *cycle* occurs in the following forms: (one cycle, two cycles, three cycles, etc.), it has a special meaning in SVA, because there are two assertion types: *immediate assertions* which are based on event semantics and *concurrent assertions* which are based on clock semantics. Therefore we must treat the word "cycle" differently to any other events. For example, "three cycles" in (139) means that the main clause "Awvalid must remain high" will be evaluated for the specified number of clocking cycles. Recall that TempCNL will treat certain words in a special way. Hence, we would prefer to interpret "three cycles" as $|3\rangle$ where we omit the word *cycles* from any TPL^+ formula if it occurs with a cardinal number.

The semantics of sentences (140)–(141) in TPL^+ are:

(142) $|3\rangle_* [\![high(Awvalid)]\!]$

(143) $\{ACK^l\}_> \langle occur(Awid\rangle)\top$

(144) $[high(Acvalid)]_> \langle Awvalid^{\Leftarrow}\rangle\top$.

From Definitions 4.4.4 and 4.4.5, we see that the truth-conditions of the above TPL^+-formulas correspond exactly to ITL formulas as shown below, respectively

(145) $\forall J(J \subseteq [start(I), start(I)+3] \rightarrow high(Awvalid, J))$

(146) $\imath J_2(ACK(J_2) \wedge J_2 \subseteq I \wedge \forall J_1(ACK(J_1) \rightarrow \neg J_1 \subseteq fin(J_2, I)),$
$\exists J_0(J_0 \subseteq fin(J_2, I) \wedge high(Awid, J_0)))$.

(147) $\forall J_1(high(Acvalid)(J_1) \wedge J_1 \subseteq I \rightarrow$
$\exists J_0(J_0 \subseteq [end(J_0)+1, end(J_0)+1] \wedge$
$(Awvalid([start(J_0), start(J_0)]) \Leftrightarrow Awvalid([start(J_0)-1, start(J_0)-1]))))$.

We derive the ITL formula (145) from the TPL^+-formula (142) using the clauses (8) and (4) in Definition 4.4.5, the ITL formula (146) from the TPL^+-formula (143) using the clause (3) in Definition 4.4.4 and the clauses (12) and (1) in Definition 4.4.5. Moreover, We derive the ITL formula (147) from the TPL^+-formula (144) using the clauses (9) and (3) in Definition 4.4.5.

Let us explain the interpretations of the above ITL formulas. Formula (145) asserts that, within the given temporal context I, every interval J includes the occurrence of *Awvalid* being high and J is contained in $[start(I), start(I)+3]$. The last interpretation is (146) which asserts that, within the given temporal context I, there is an interval J_2 over which ACK begins and ends after all other ACKs; after the interval J_2, there is an interval J_0 includes the occurrence of *Awid*. The meaning of formula (147) asserts that, within the given temporal context I, immediately after every interval J_1 over which *Acvalid* is high, there exists a time point J_0 over which *Awvalid* is true if and only if *Awvalid* held at the previous time point $[start(J_0) - 1, start(J_0) - 1]$.

TPL^+ formulas are evaluated over discrete time intervals. However, *SVA* (closely related to *LTL*) employs predicates evaluated at a single point in time and does not allow arbitrary quantification. So it is not obvious how we can map TPL^+ to *SVA*. This problem is shown in Chapter 5.

4.5 Grammar

This section shows the TPE grammar which is used to extract TPL^+ formulas from TempCNL sentences. The TPE grammar uses a depth-first top-down parser as discussed in Section 2.2. The TPE grammar is a context-free grammar that combines typed logic with lambda abstraction to obtain TPL^+ formulas. This method, defined in Montague (1974), has been used to build semantic representations for various fragments of English. However, we use Montague semantics to generate TPL^+ formulas from strings of symbols as described previously in Section 2.1. This means that those strings in our grammar will not have full meanings until TPL^+ formulas are extracted.

In this section, we first discuss the importance of adding syntactic restrictions on our grammar to generate TPL^+ formulas. Then, we provide the grammar rules of sentence structure, temporal preposition phrase with nominal complements, temporal preposition phrase with clausal complements, and coordination categories separately.

4.5.1 Syntactic Restrictions

This section shows how and why we add some syntactic restrictions to certain phrase categories in the TPE grammar. We first show a sample of our grammar and how we generate the phrase structures of some sentences using our grammar. Then, we discuss the necessity of adding syntactic restrictions on the grammar.

Let us show a sample of our grammar rules. Figure 4.5.1 presents a grammar rules that written in SWI-prolog. As shown in Figure 4.5.1, the head of the TPE grammar is the `ip1` category. The term `ip` refers to a phrase that is headed by a tense or an auxiliary. We usually represent a sentence as an inflectional phrase. The `ip1` category consists of another inflectional phrase `ip0`. We use a different number for the embedded inflectional phrase to avoid left recursive grammar rules. We use the same method for noun phrase categories (e.g. `np1` and np0). The structure of `ip1` also consists of the `ip0` category, followed by a temporal preposition phrase `tpp`.

The grammar rules, in Figure 4.5.1, are built entirely using a CFG formalism. A CFG is similar to a *definite clause grammar* (DCG) but is less powerful since it has no variables in the productions. However, if G is a DCG but variables rang over finite sets, there exists a CFG G' such that $L(G) = L(G')$. Thus, our grammar has four variables: `Q`, `M`, `Num` and `V`. These variables are located in the syntactic features `qclause`, `mclause`, `type` and `symbol`, respectively. We use the first two variables to give syntactic restrictions on the complements and main clauses for some temporal prepositions, receptively. The variable `Q` ranges over `forall`, `exists`, `init_exists` and `def` tags which denote universal, existential, initial existential and definite quantifications. On the other hand, the variable `M` ranges over `forall`, `exists` and `init_exists` tags which denote universal, existential and initial existential quantifications. Moreover, we use the variable `Num` to enforce subjects and verbs to agree with one another in number (singular or plural). These variables range over finite sets of tags. Therefore, using variables in our grammar does not take us out of CFG. For linguistic processing, the variable `V` in the `symbol` feature is used in

```
/*==============================================================
   Phrase rules
===============================================================*/
ip1 -->  ip0([mclause:exists]).
ip1 -->  ip0([mclause:forall]).
ip1 -->  ip0([mclause:M]), tpp([mclause:M]).
ip0([mclause:M])--> np0([num:Num]),
                    ibar([num:Num,mclause:M]).
tpp([mclause:M]) -->  tpn([qclause:Q,mclause:M]),
                      tnp0([qclause:Q,num:_]).
np0([num:Num])-->  pn([num:Num]).
tnp0([qclause:Q,num:Num]) --> det([type:Q]),
                             tnbar0([type:Q,num:Num]).
tnbar0([type:def,num:Num]) --> tAdj, tn([num:Num]).
tnbar0([type:_,num:Num]) --> tn([num:Num]).
ibar([num:Num,mclause:M])--> aux([num:Num]),
                             vp([symbol:[],mclause:M]).
ibar([num:Num,mclause:M])--> i([symbol:V,num:Num]),
                             vp([symbol:V,mclause:M]).
vp([symbol:V,mclause:M])--> v([symbol:V,mclause:M]).
v([symbol:V,mclause:M])-->  iv([symbol:V,mclause:M]).
v([symbol:V,mclause:M])-->  iv_adj([symbol:V,mclause:M]), adj.

/*==============================================================
   Closed Class Lexicon
===============================================================*/
det([type:def]) --> [the].
det([type:forall])--> [every].
det([type:exists])--> [a].
aux([num:sg])--> [is].
tAdj --> [first].
tAdj --> [last].
tpn([qclause:def,mclause:exists])-->  [by].
tpn([qclause:def,mclause:forall])-->  [until].
/*==============================================================
   Open Class Lexicon
===============================================================*/
tn([num:sg])--> [transaction].
pn([num:sg])--> ['Awid'].
pn([num:sg])--> ['Awvalid'].
adj --> [high].
adj --> [low].
i([symbol:[remain],num:sg]) -->   [remains].
iv([symbol:[],mclause:exists]) -->  [asserted].
iv_adj([symbol:[remain],mclause:forall]) --> [ ].
```

Figure 4.5.1: Simple grammar rules for TempCNL.

our grammar for a simple verb movement. The variable `V` stores the base form of the verb in the `i` category and moving it to the appropriate position where we can extract its meaning. Since we use only a simple verb movement, we still can build our grammar rules in a CFG formalism.

Let us start with two simple examples of this structure in English:

(148) Awid is asserted.

(149) Awvalid remains high.

We use the grammar rules in Figure 4.5.1 to generate the phrase structures of the above two sentences as shown in Figure 4.5.2. In Figure 4.5.2, in the top phrase structure, the `ip0` category is tagged with existential quantification since the verb "asserted" is an event verb while in the bottom phrase structure the `ip0` category is tagged with universal quantification since the verb "remains" is an state verb. The process of selecting the quantification type of these verbs is discussed in Section 4.3.2.

Note that, in the bottom phrase structure in Figure 4.5.2, the verb "remain" is moved to the `iv-adj` category and then we assign universal quantification for it. However, if the `vp` category combines with the `aux` category, the value in the `symbol` feature will be empty (which denoted by []), because there is no verb movement that required in this rule. For example, in the top phrase structure in Figure 4.5.2, the value in the `symbol` feature is bound by the [] since there is no verb movement in the given sentence.

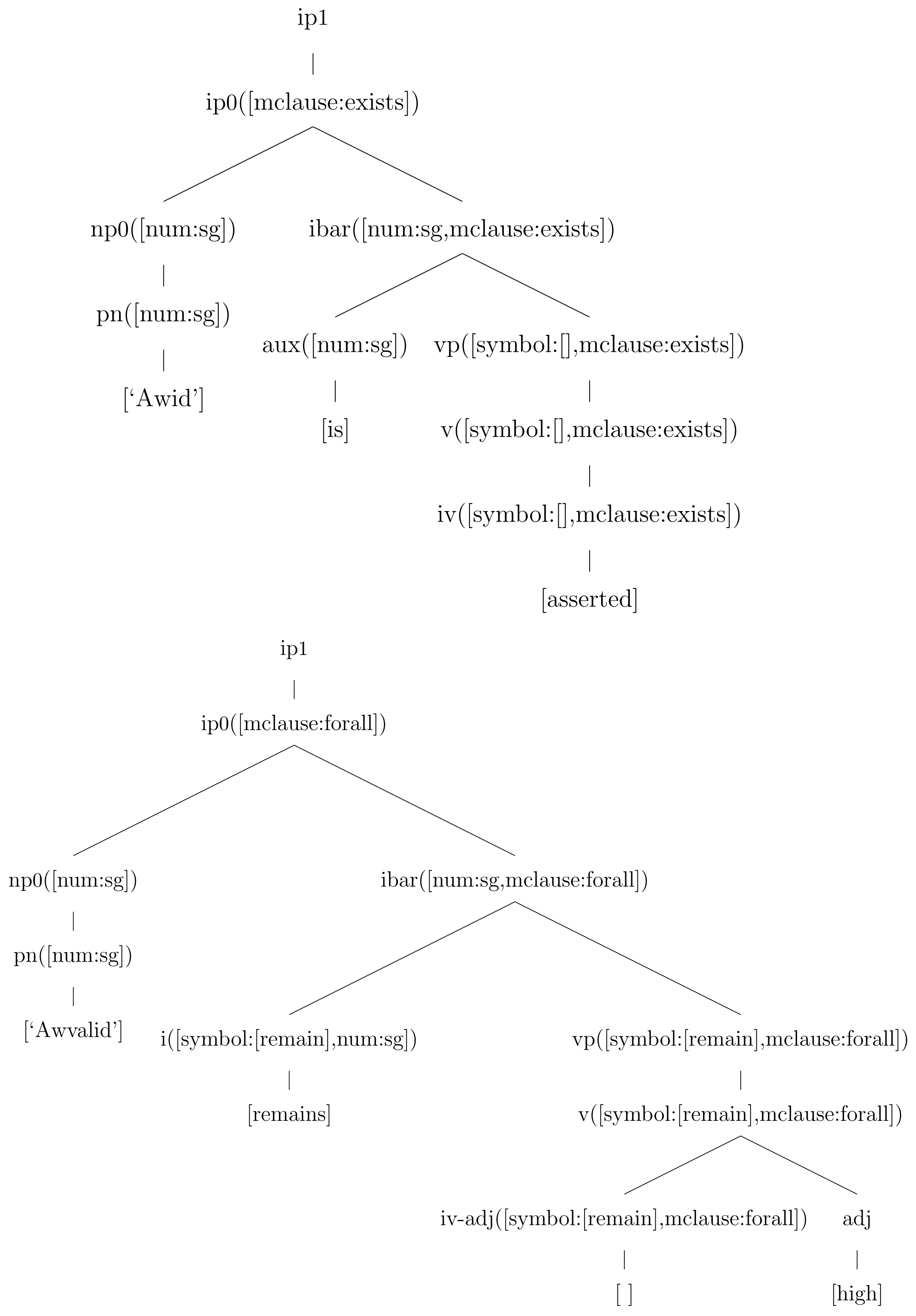

ip1
ip0([mclause:exists])
np0([num:sg])
ibar([num:sg,mclause:exists])
pn([num:sg])
['Awid']
aux([num:sg])
[is]
vp([symbol:[],mclause:exists])
v([symbol:[],mclause:exists])
iv([symbol:[],mclause:exists])
[asserted]
ip1
ip0([mclause:forall])
np0([num:sg])
ibar([num:sg,mclause:forall])
pn([num:sg])
['Awvalid']
i([symbol:[remain],num:sg])
[remains]
vp([symbol:[remain],mclause:forall])
v([symbol:[remain],mclause:forall])
iv-adj([symbol:[remain],mclause:forall])
[ ]
adj
[high]

Figure 4.5.2: The phrase structures of sentences (148) and (149).

Let us now explain briefly why we give some syntactic restrictions to certain phrase categories using variable such `Q` and `M` in the grammar rule in Figure 4.5.1. Basically, we apply these syntactic restrictions to our grammar rules to extract TPL^+ formulas based on the TPL^+ semantics given in Section 4.4. This is because there are some phrase categories, such as temporal prepositions and the adjectives *first* and *last*, that require additional restrictions on other phrase categories.

We begin by introducing the use of the variable `Q` in the `type` feature in the rules at Figure 4.5.1. One of the main reasons for using the variable `Q` is to impose syntactic restrictions on phrases taking the adjectives *first* and *last* – specifically, to require them to combine with the definite article. This is due to the fact that it is ordinarily difficult to combine the adjectives *first* and *last* with a universal or existential article such as "a first" or "every first". Of course, there are some English sentences such as the following:

(150) You can have one last chance;

(151) Every first attempt at the test resulted in failure;

where the adjectives *first* and *last* come with existential and universal articles. However, the phrase "one last chance" in sentence (150) is used here in a non-literal way because it cannot be another chance unless if they occur simultaneously, which is impossible. On the other hand, the phrase "every first attempt" involves a repeated sequence of temporal contexts within which the adjectives *first* and *last* are interpreted. Note that these types of phrases are not common in natural language protocol specifications. Thus, we do not include them in our TempCNL. To show how we apply the restriction on the adjectives *first* and *last*, see the phrase structure of (156) in Figure 4.5.10.

Our second reason for using additional syntactic restrictions is that some temporal prepositions enforce their complements and main clauses to be restricted with particular quantifications. For example, consider the following sentences:

(152) ? Awvalid remains asserted by the transaction.

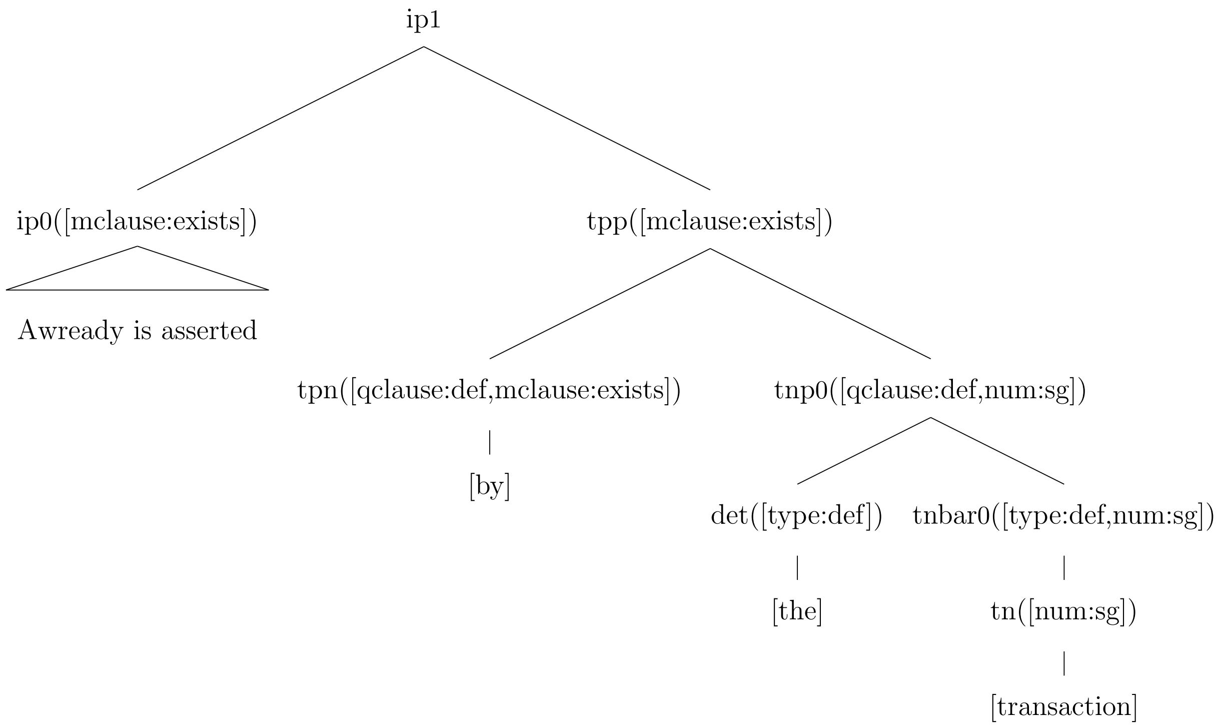
(153) Awvalid is asserted by the transaction.

(154) Awvalid remains high until the transaction.

The temporal prepositions *until* and *by* require their complements to be definitely quantified as explained previously in examples (83) and (84) in Section 3.3.1. Moreover, the temporal preposition *by* naturally forces its main clause to take existential quantification while the temporal preposition *until* forces its main clause to take universal quantification. Figure 4.5.3 shows how we generate the phrase structures for sentences (153) and (154) using the grammar rules in Figure 4.5.1.

As shown in Figure 4.5.3, both temporal prepositions *until* and *by* have two syntactic features called `qclause` and `mclause`. Each temporal preposition has both features to store the value of quantification type of its complement in the `qclause` feature and its main clause in the `mclause` feature. These features are defined in TempCNL lexicon in Section 4.3.1.

Moreover, in the grammar rules in Figure 4.5.1, if both `ip0` and `tpp` categories have the same value of quantification type in the `mclause` feature, then the grammar rules generate the phrase structures for the corresponding sentences such as (153) and (154); otherwise, no result will be computed. For example, (152) is considered as an ungrammatical sentence in the TPE grammar since the temporal prepositions *by* forces its main clause to take existential quantification while its main clause here is restricted with the universal quantification. We discuss the use of these variables in detail in the upcoming sections.



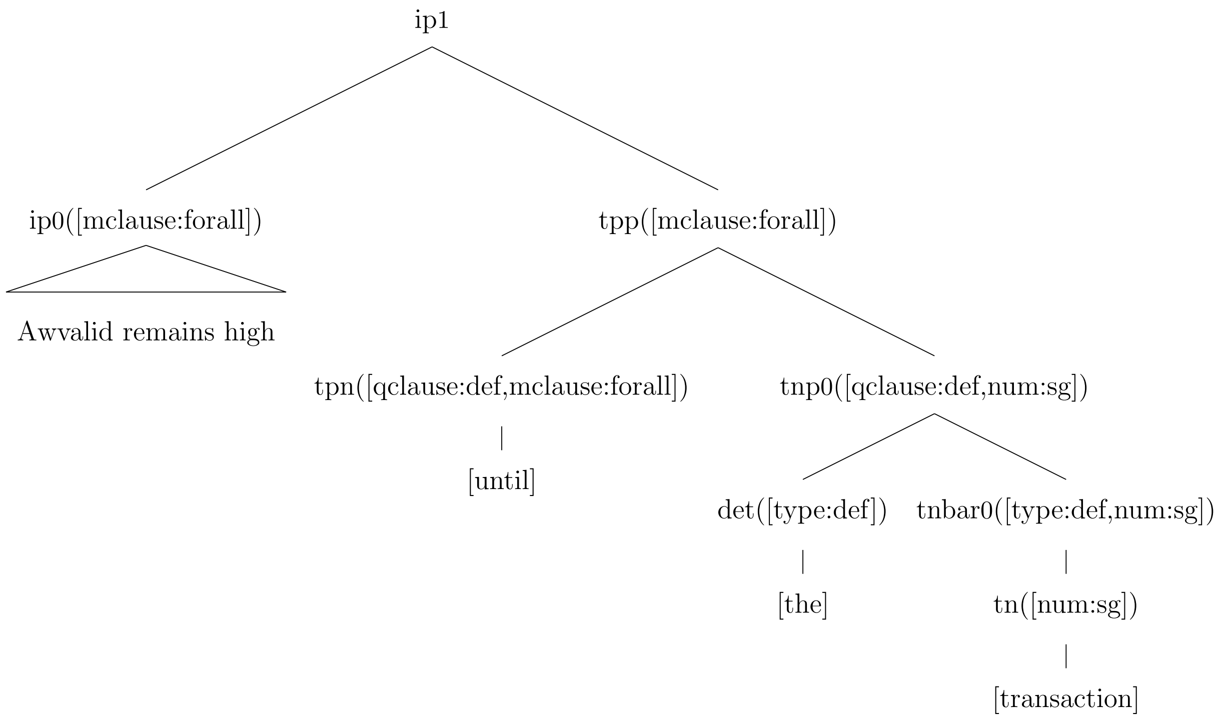


Figure 4.5.3: The structure of sentences (153) and (154), respectively.

4.5.2 Sentence Structure

Now let us show how we extract TPL^+ formulas from simple sentences. We show the grammar rules for sentence, temporal noun, non-temporal noun and predicate categories respectively. Then, we illustrate the process for extracting TPL^+ formulas from each category using phrase structures.

We begin with writing the grammar rules for the `ip2`, `ip1`, `ip0` categories as follows:

```
ip2([sem:IP])-->  ip1([sem:IP]).
ip2([sem:TPP@IP])-->  tpps([sem:TPP]),ip1([sem:IP]).
ip2([sem:TPP@IP])-->  ip1([sem:IP]), tpps([sem:TPP]).
ip1([sem:TPP@IP])-->  tpp([mclause:M,sem:TPP]),
                      ip0([mclause:M,sem:IP]).
ip1([sem:TPP@IP])-->  ip0([mclause:M,sem:IP]),
                      tpp([mclause:M,sem:TPP]).
ip1([sem:IP])-->  ip0([mclause:exists,sem:IP]).
ip1([sem:IP])-->  ip0([mclause:forall,sem:IP]).
ip0([mclause:M,sem:IBar@NP])-->  np0([num:Num,sem:NP]),
                                 ibar([num:Num,mclause:M,sem:IBar]).
```

Figure 4.5.4: Grammar rules for inflectional phrase.

The head of the TPE grammar is the `ip2` category. In the above rules, we add a semantic feature called `sem` to store the corresponding string in every category. The following rule is a part of the rules in Figure 4.5.4, and it is interpreted as follows: a sentence consists of a `tpp` category, followed by a `ip0` category. If the `sem`'s value of the `tpp` category is `TPP` and the `sem`'s value of the `ip0` category is `IP`, then the `sem`'s value of the `ip1` category will be the result of `TPP@IP`, which means we apply `IP` to `TPP`. Other rules in Figure 4.5.4 are interpreted similarly.

```
ip1 ([sem:TPP@IP]) --> tpp ([mclause:M,sem:TPP]),
                       ip0 ([mclause:M,sem:IP]).
```

Figure 4.5.5: A sample rule for the `ip1` category.

The @-operator usually is used to combine semantic representations while parsing a sentence, then β-converting the result in a subsequent post-processing step. However, we use the @-operator to show the order of applying the function to the argument statements.

In practice, we compute the `sem`'s value of each category directly in the grammar rules as shown below with the previous rules:

```
ip1([sem:IP1]) --> tpp([mclause:M,sem:TPP]),
                   ip0([mclause:M,sem:IP0]),
                   {var_replace(TPP,TPP1),beta(TPP1@IP0,IP1)}.
```

Figure 4.5.6: Computing the `sem`'s value of the `ip1` category using the predicates `var_replace` and `beta`.

Here, the predicate `var_replace` is used to perform α-conversion and the predicate `beta` is used to implement β-reduction, assuming that there are no variable clashes. These two predicates are called in each rule to compute the `sem`'s value. Note that the rules that consist of one category such as the following rule:

```
ip2([sem:IP])-->  ip1([sem:IP]).
```

where the left-hand side has only one category that does not require any computation. Note that we omit to write the predicates `var_replace` and `beta` in our grammar for simplicity and write instead the @-operator as shown with the rule in Figure 4.5.5 since it is clear to the reader how this part can be done from the given rule in Figure 4.5.6.

As previously mentioned, in the grammar rules at Figure 4.5.4, the variable `M` ranges over the tags `forall`, `exists` and `init_exists`, which denote universal, existential and initial existential quantifications which are used in our grammar to give syntactic restrictions on the main clauses for some temporal prepositions.

Let us start with an example of a simple sentence in English:

(155) Awvalid is asserted.

To obtain the TPL^+ formula for (155), Figure 4.5.7 shows how to assign the phrase structure using the grammar rules in Figure 4.5.8. (Note that these grammar rules are parts of the grammar rules that presented in Figures 4.5.4, 4.5.12 and 4.5.14.)

As shown in Figure 4.5.7, we produce TPL^+ formula "$\langle assert(Awvalid)\rangle\top$" for sentence

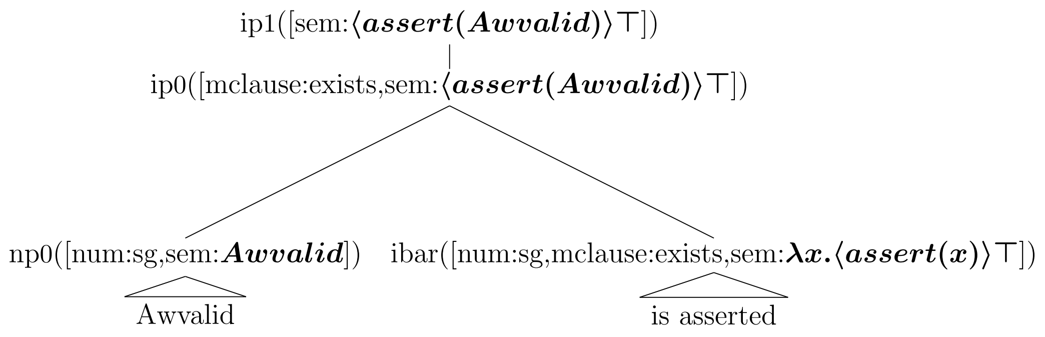

ip1([sem:⟨assert(Awvalid)⟩⊤])
ip0([mclause:exists,sem:⟨assert(Awvalid)⟩⊤])
np0([num:sg,sem:Awvalid])
ibar([num:sg,mclause:exists,sem:λx.⟨assert(x)⟩⊤])
Awvalid
is asserted

Figure 4.5.7: The phrase structure of sentence (155).

```
/*==============================================================
   Phrase structure rules
===============================================================*/
ip2([sem:IP])-->  ip1([sem:IP]).
ip1([sem:IP])-->  ip0([mclause:exists,sem:IP]).
ip0([mclause:M,sem:IBar@NP])-->  np0([num:Num,sem:NP]),
                                 ibar([num:Num,mclause:M,sem:IBar]).
np0([num:Num,sem:PN])-->  pn([num:Num,sem:PN]).
ibar([num:Num,mclause:M,sem:VP])-->
  aux([num:Num]), vp([symbol:[],mclause:M,sem:VP]).
vp([symbol:V,mclause:M,sem:VP])--> v([symbol:V,mclause:M,sem:VP]).
v([symbol:V,mclause:M,sem:IV])--> iv([symbol:V,mclause:M,sem:IV]).
/*==============================================================
   Class Lexicon
===============================================================*/
pn([num:_,sem:Awvalid]) --> ['Awvalid'].
iv([symbol:[],mclause:exists,sem:λx.⟨assert(x)⟩⊤]) --> [asserted].
aux([num:sg])--> [is].
```

Figure 4.5.8: Grammar rules for sentence (155).

(155) by replacing the string x with $Awvalid$ in $\langle assert(x)\rangle\top$. Note that the `ip0` category is tagged with existential quantification since the verb "asserted" is an event verb. Note also that the above phrase structure is not fully explained at this stage, but it will be discussed later. In particular, the top phrase structure in Figure 4.5.15 is the extended version of the phrase structure in Figure 4.5.7.

A further note on the phrase structure in Figure 4.5.7 and the grammar rules in Figure 4.5.8, we use the TPL^+ syntax in LaTeX expressions rather than the actual forms in SWI-Prolog because (1) SWI-Prolog does not recognize the TPL^+ syntax that defined in Section 4.4 and (2) we want to eliminate any confusion between the TPL^+ syntax and the different formats that are used for it in SWI-Prolog. In practice, we use the ASCII codes to represent the TPL^+ syntax in SWI-Prolog as shown in Table 4.5.1, where it provides the TPL^+ syntax with their corresponding ASCII codes in SWI-Prolog.

TPL^+ syntax	Their ASCII codes	TPL^+ syntax	Their ASCII codes
$\lambda x.x$	`lbd(x,x)`	$\neg\phi$	$\sim\phi$
$\langle e\rangle$	`exists(e)`	$\phi_{<}\phi'$	ϕ `before` ϕ'
$\|e\rangle$	`init_exists(e)`	$\phi_{>}\phi'$	ϕ `after` ϕ'
$[e]$	`forall(e)`	$\phi_{=}\phi'$	ϕ `during` ϕ'
$\{e\}$	`def(e)`	$\phi_{<+}\phi'$	ϕ `until_after` ϕ'
e^f	`f(e)`	$\phi_{*}\phi'$	ϕ `repeat` ϕ'
e^l	`l(e)`	$\phi \wedge \phi'$	ϕ `&` ϕ'
$e^{\Leftarrow}$	`e<<-`	$\phi \vee \phi'$	ϕ `v` ϕ'

Table 4.5.1: The TPL^+ syntax with their corresponding ASCII codes in SWI-Prolog.

In this chapter, we will use the LaTeX expressions for TPL^+ syntax in our grammar rules rather than the actual forms in Prolog. However, in Appendix B, we use the ASCII codes in our grammar rules.

Let us now explain the internal structure of the `ip0` category in detail.

4.5.2.1 Noun Phrases

English noun phrases are usually taken to include temporal and non-temporal nouns. Our grammar will treat them separately. We follow the same approaches in Pratt and Francez (1997, 2001) mentioned previously in Section 3.3. This distinction will play an important role when we extract the semantics of temporal preposition phrases, since temporal nouns can help us to distinguish between temporal prepositions and other types of prepositions (we discuss this topic, in Section 4.5.3).

Let us write the grammar rules for temporal noun phrases as given in Figure 4.5.9.

We now show how to extract the semantics of temporal noun phrases using our rules. Consider the following examples:

(156) every reset

(157) the first acknowledgement

```
/*=============================================================
   Temporal Noun Phrase rules
==============================================================*/
tnp0([qclause:D,num:Num,sem:Det@TnBar]) -->
    det([type:D,sem:Det]), tnbar0([type:D,num:Num,sem:TnBar]).
tnp0([qclause:init_exists,num:Num,sem:NumS]) -->
    num([sem:NumS]), tnbar0([type:_,num:Num,sem:_]).
tnbar0([type:def,num:Num,,sem:Tadj@Tn]) -->
    tAdj([sem:Tadj]), tn([num:Num,sem:Tn]).
tnbar0([type:_,num:Num,sem:Tn]) --> tn([num:Num,sem:Tn]).
/*=============================================================
   Open Class Lexicon
==============================================================*/
tn([num:pl,sem:cycle])--> [cycles].
tn([num:sg,sem:reset])--> [reset].
tn([num:sg,sem:ack])--> [acknowledgement].
num([sem:|3⟩]) --> ['3'].
num([sem:|16⟩]) --> ['MaxWaits'].
/*=============================================================
   Closed Class Lexicon
==============================================================*/
det([type:forall,sem:λx.[x]]) --> [every].
det([type:exists,sem:λx.⟨x⟩]) --> [some].
det([type:def,sem:λx.{x}]) --> [the].
tAdj([sem:λx.x^f]) --> [first] .
tAdj([sem:λx.x^l]) --> [last].
```

Figure 4.5.9: Grammar rules for temporal noun phrases.

where the nominal expressions *reset* and *acknowledgement* are temporal nouns since they refer to event-types. To extract the strings of noun expressions in (156) and (157), we assign the phrase structures using the rules in Figure 4.5.9 as shown in Figure 4.5.10.

As can be seen in Figure 4.5.10, λx takes the string x and substitute it with the string of the `tnbar0` category in $[x]$. Moreover, the top phrase shows that the `tnbar0` category is tagged with universal quantification since it combines with the `det` category that is tagged with universal quantification, whereas the bottom phrase shows that the `tnbar0` category is tagged with definite quantification not only because it combines with the `det` category that has a definite article, but also because the `tnbar0` category has the order-denoting adjective *first* which naturally enforces its determiner to be a definite article as explained previously in Section 4.5.1. Note that the strings of `tnp0` (such as $\{ack^f\}$ and $[reset]$) are taken to be the first argument of the modal operators $<$, $<+$, $>$ and $=$ in the TPL^+ language.

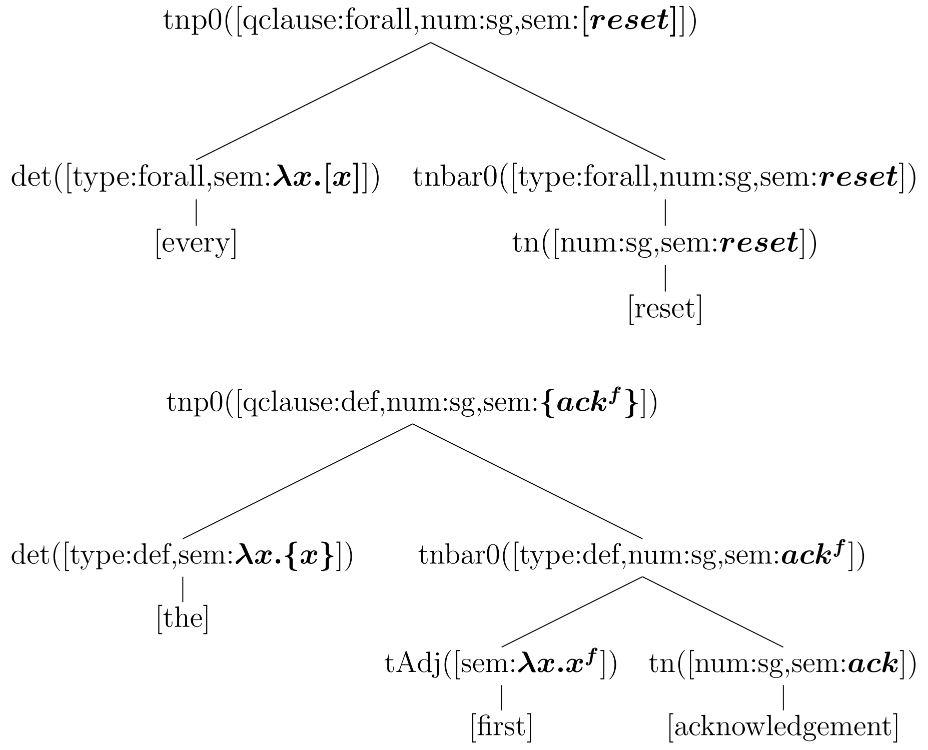

tnp0([qclause:forall,num:sg,sem:[reset]])
det([type:forall,sem:λx.[x]])
[every]
tnbar0([type:forall,num:sg,sem:reset])
tn([num:sg,sem:reset])
[reset]
tnp0([qclause:def,num:sg,sem:{ack^f}])
det([type:def,sem:λx.{x}])
[the]
tnbar0([type:def,num:sg,sem:ack^f])
tAdj([sem:λx.x^f])
[first]
tn([num:sg,sem:ack])
[acknowledgement]

Figure 4.5.10: The phrase structures of temporal noun expressions in (157) and (156).

Let us now turn to how we extract the string of the `num` category when it combines with the word *cycles* such as in (158) and (159) using the rules we have set out.

(158) 3 cycles

(159) MaxWaits cycles.

In (158), "3 cycles" has the cardinal number 3 which describes the number of repetitions that the cycle generates as mentioned with example (139) in Section 4.4. In (159), MaxWaits is a constant and its value is 16 as shown in our grammar rules. Thus, we assign the phrase structures using the rules in Figure 4.5.9 as shown in Figure 4.5.11.

In both phrase structures in Figure 4.5.11, we ignore the `sem`'s value of the `tnbar0` category since the word *cycles* has no effect on the TPL^+ semantics if it occurs with a cardinal number as explained earlier with example (139). From the grammar rules in Figure 4.5.9, We employ the rule

```
tnp0([qclause:init_exists,num:Num,sem:NumS]) -->
    num([sem:NumS]), tnbar0([type:_,num:Num,sem:_]).
```

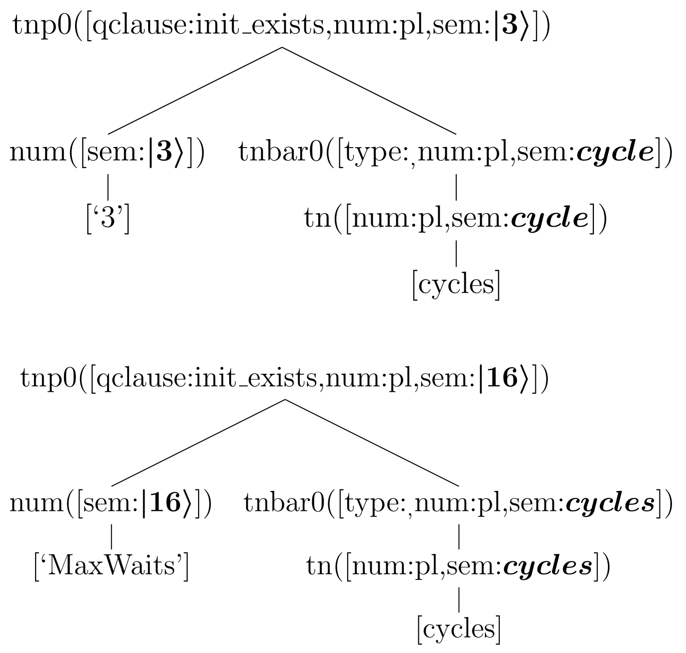

tnp0([qclause:init_exists,num:pl,sem:|**3**⟩])
num([sem:|**3**⟩])
tnbar0([type:,num:pl,sem:***cycle***])
[‘3’]
tn([num:pl,sem:***cycle***])
[cycles]
tnp0([qclause:init_exists,num:pl,sem:|**16**⟩])
num([sem:|**16**⟩])
tnbar0([type:,num:pl,sem:***cycles***])
[‘MaxWaits’]
tn([num:pl,sem:***cycles***])
[cycles]

Figure 4.5.11: The phrase structures of temporal noun expressions in (158) and (159).

to take `NumS` as the `sem`'s value of the `num` category and ignore the `sem`'s value of the `tnbar0` category. We also tag the `tnp0` category with initial existential quantifier using the same grammar rule. This tag type enforces the strings of the `tnp0` categories having the `num` category (such as $|3\rangle$ and $|16\rangle$) to combine only with the modal operator $*$ as its first argument. As defined in Section 4.4, the modal operator $*$ in $|n\rangle_* \psi$ denotes the repetition of ψ a given number of times. Note that $|3\rangle$ and $|16\rangle$ are strings which do not mean here as TPL^+ formulas. For a complete example of this kind of construction, see the top phrase structure in Figure 4.5.23. The rest of the grammar rules for temporal noun phrases are given in Appendix B.

Now, let us write the grammar rules for the non-temporal noun phrases as given in Figure 4.5.12. The remaining grammar rules for non-temporal noun phrases are given in Appendix B.

In our grammar rules in Figure 4.5.12, we do not have a rule that allows a noun phrase including non-temporal quantification. Therefore, we cannot say every signal or some signal of a certain kind in TempCNL. This is because those articles are considered as non-temporal quantification whereas TPL^+, *LTL* and *SVA* are propositional languages—all

```
/*=============================================================
   Non-temporal noun phrase
=============================================================*/
np0([num:Num,sem:PN]) -->
     det([type:def,sem:_]), pn_def([num:Num,sem:PN]).
np0([num:Num,sem:PN])-->  pn([num:Num,sem:PN]).
/*=============================================================
   Open Class Lexicon
=============================================================*/
pn_def([num:_,sem:Awready]) --> ['Awvalid'].
pn_def([num:_,sem:Acvalid]) --> ['Awid'].
pn([num:_,sem:Awvalid]) --> ['Awvalid'].
pn([num:_,sem:Awid]) --> ['Awid'].
```

Figure 4.5.12: Grammar rules for non-temporal noun phrases.

quantification is temporal. Thus, they cannot cope with non-temporal quantification as explained previously in Section 4.1. Therefore, such these noun phrases should not be processed by our grammar because their interpretations will take us outside the TPL^+ language. The non-temporal noun phrases in TempCNL can be either single names such as "Awvalid" or single names that allow a preceeding definite article such as "the Awid".

Let us use the above rules to extract the semantics of a non-temporal noun phrase. Consider the following examples:

(160) the Awid

(161) Awvalid

where we assign the phrase structures of the above noun expressions using the rules in Figure 4.5.12 as shown in Figure 4.5.13.

In Figure 4.5.13, the `sem`'s value of the noun phrase "the Awid" will be only "Awid" since definite article comes optional with the signal names. Therefore, we ignore the `sem`'s value of the `det` category using the rules in Figure 4.5.12. Note that we enforce the `det` category to have only definite article by using the `def` tag in the same rule. Later, the `sem`'s values of the proper nouns "Awid" and "Awvalid" will be applied with the `sem`'s value of the predicate phrase. In particular, the `sem`'s value of predicate phrase `ibar` takes the `sem`'s value of non-temporal noun phrases as its arguments. We will see how this can

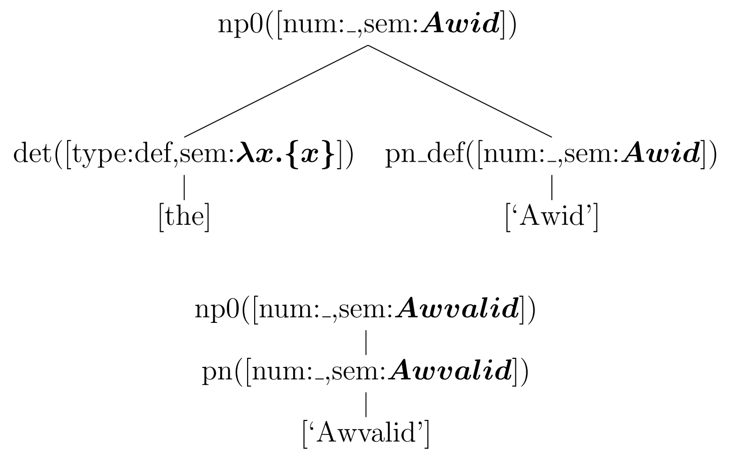

np0([num:_,sem:$\boldsymbol{Awid}$])
det([type:def,sem:$\boldsymbol{\lambda x.\{x\}}$])
pn_def([num:_,sem:$\boldsymbol{Awid}$])
[the]
['Awid']
np0([num:_,sem:$\boldsymbol{Awvalid}$])
pn([num:_,sem:$\boldsymbol{Awvalid}$])
['Awvalid']

Figure 4.5.13: The phrase structures of noun expressions in (160) and (161).

be computed in Section 4.5.2.2.

4.5.2.2 Predicate Phrase

The predicate phrase belongs to the `ibar` category, which usually has the following subcategories: an inflection (`i`), an auxiliary verb (`aux`), a negated phrase (`negP`) and a verb phrase (`vp`). Those categories tell us when the events are happening and whether they extend in time or occur at an instantaneous time as explained in Section 4.3.

We begin with the grammar rules for the predicate phrase as given in Figure 4.5.14.

In our grammar rules in Figure 4.5.14, auxiliary verbs, such as *must* and *is*, are not included in TPL^+ since, in this thesis, we deal with natural language requirements where these types have no major effects on the interpenetration of formal specification, namely *SVA*. Thus, we only focus on the main verbs in this domain.

Moreover, in our grammar in Figure 4.5.14, the bracket-pairs $\langle\ \rangle$, $[\![\ ]\!]$, $|\ \rangle$ denote existential , universal and initial existential quantifiers, respectively, and the $\langle x^{\Leftarrow}\rangle\top$ indicates that x holds at the time of evaluation if and only if it held at the previous time point (they are introduced in the TPL^+ language as base cases). After we assign these forms to their arguments, they will be taken as arguments of the modal operators $\{<,\ <+,\ >,\ =,\ *\}$

```
/*==============================================================
   Predicate Phrases
==============================================================*/
ibar([num:Num,mclause:M,sem:NegP])-->
   aux([num:Num]), negP([symbol:[],mclause:M,sem:NegP]).
ibar([num:Num,mclause:M,sem:VP])-->
  aux([num:Num]), vp([symbol:[],mclause:M,sem:VP]).
ibar([num:Num,mclause:M,sem:VP])-->
   i([symbol:V,num:Num]), vp([symbol:V,mclause:M,sem:VP]).
negP([symbol:V,mclause:NewM,sem:Neg@VP])-->
   neg([sem:Neg]), vp([symbol:V,mclause:M,sem:VP]),
   {quantifier_modification(M,NewM)}.
vp([symbol:V,mclause:M,sem:VP])-->
   extraAux, v([symbol:V,mclause:M,sem:VP]).
vp([symbol:V,mclause:M,sem:VP])--> v([symbol:V,mclause:M,sem:VP]).
v([symbol:V,mclause:M,sem:IV])--> iv([symbol:V,mclause:M,sem:IV]).
v([symbol:V,mclause:M,sem:Adj]) -->
  iv_adj([symbol:V,mclause:_,sem:_]), adj([type:M,sem:Adj]).
v([symbol:V,mclause:M,sem:IV@Adj])-->
  iv_adj([symbol:V,mclause:M,sem:IV]), adj([sem:Adj]).
v([symbol:V,mclause:M,sem:TV@NP])-->
   tv([symbol:V,mclause:M,sem:TV]), np0([num:_,sem:NP]).
/*==============================================================
   Open Class Lexicon
==============================================================*/
adj([sem:λx.high(x)]) --> [high].
adj([sem:λx.low(x)]) --> [low].
i([symbol:[remain],num:sg]) -->   [remains].
i([symbol:[go],num:sg]) -->   [goes].
iv([symbol:[],mclause:exists,sem:λx.⟨assert(x)⟩⊤]) --> [asserted].
iv([symbol:[],mclause:init_exists,sem:λx.|assert(x)⟩⊤]) --> [asserted].
iv_adj([symbol:[remain],mclause:forall,sem:λpλx.⟦p(x)⟧]) --> [ ].
iv_adj([symbol:[],mclause:forall,sem:λpλx.⟦p(x)⟧]) --> [remain].
iv_adj([symbol:[go],mclause:exists,sem:λpλx.⟨p(x)⟩⊤]) --> [ ].
iv_adj([symbol:[go],mclause:init_exists,sem:λpλx.|p(x)⟩⊤]) --> [ ].
/*==============================================================
   Closed Class Lexicon
==============================================================*/
aux([num:sg])--> [is].
aux([num:pl])--> [are].
aux([num:_])--> [must].
neg([sem:λpλx.¬p(x)]) --> [not].
neg([sem:λpλx.¬p(x)]) --> [never].
extraAux  -->   [be].
adj([type:exists,sem:λx.⟨x^⇐⟩⊤]) --> [stable].
adj([type:init_exists,sem:λx.⟨x^⇐⟩⊤]) --> [stable].
```

Figure 4.5.14: Grammar rules for the predicate phrase.

in the TPL^+ language.

Another feature of our grammar rules in Figure 4.5.14 is that any verb which has existential quantification can also have initial existential quantification, such as the verbs “asserted” and “go”. Does this mean that we have ambiguity in our grammar? No, because (i) if we have a sentence that only consists of a noun phrase, followed by a predicate phrase, then we only allow this sentence to be either universally or existentially quantified by using the following rules from Figure 4.5.4:

```
ip1([sem:IP])-->  ip0([mclause:exists,sem:IP]).
ip1([sem:IP])-->  ip0([mclause:forall,sem:IP]).
```

and (ii) if we have a sentence that consists of a temporal preposition phrase, followed by a sentence or vice versa, then the grammar rule for that temporal preposition will determine whether its main clause requires initial existential quantification or not. For example, when the complements of *after* and *once* have universal quantification, then their main clauses must only take initial existential quantification. However, when the complement of *in* has universal quantification and the complement of *by* has definite quantification, then their main clauses must only take existential quantification. Giving two meanings for each verb will not cause ambiguity since every preposition only takes one of those meanings but not both of them. We shall see that in detail in Section 4.5.3.

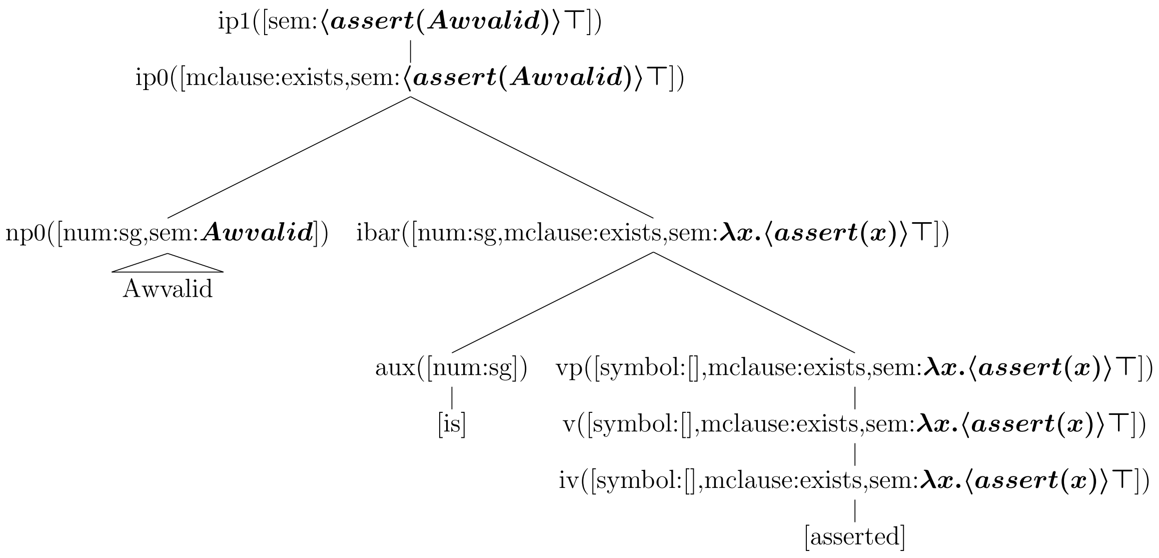
Let us use the rules in Figure 4.5.14 to extract the semantics of the predicate phrase in the following sentences:

(162) Awvalid is asserted

(163) Awvalid remains high.

We assign the phrase structures of (162) and (163) as shown in Figure 4.5.15.

As shown in Figure 4.5.15, the verb “asserted” is assigned to existential quantification $\langle\ \rangle\top$ while the verb “remains” is assigned to universal quantification $[\![\ ]\!]$. The figure shows that


ip1([sem:⟨assert(Awvalid)⟩⊤])
ip0([mclause:exists,sem:⟨assert(Awvalid)⟩⊤])
np0([num:sg,sem:Awvalid])
Awvalid
ibar([num:sg,mclause:exists,sem:λx.⟨assert(x)⟩⊤])
aux([num:sg])
[is]
vp([symbol:[],mclause:exists,sem:λx.⟨assert(x)⟩⊤])
v([symbol:[],mclause:exists,sem:λx.⟨assert(x)⟩⊤])
iv([symbol:[],mclause:exists,sem:λx.⟨assert(x)⟩⊤])
[asserted]

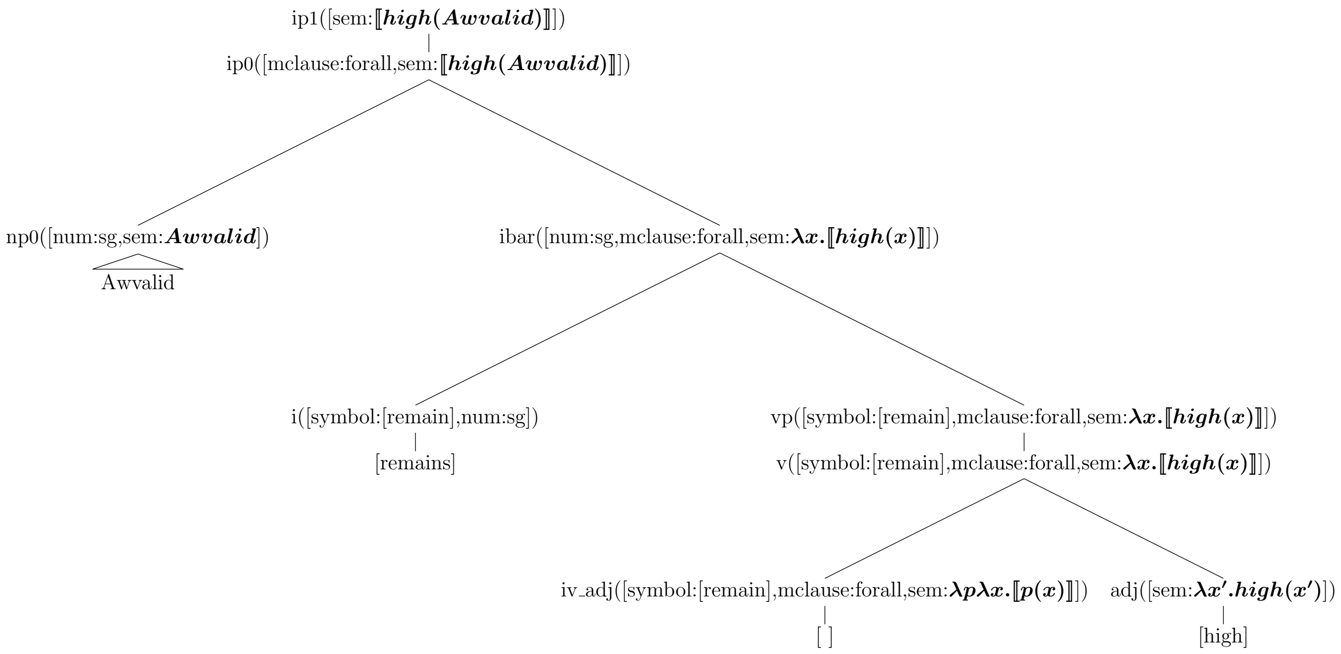

ip1([sem:⟦high(Awvalid)⟧])
ip0([mclause:forall,sem:⟦high(Awvalid)⟧])
np0([num:sg,sem:Awvalid])
Awvalid
ibar([num:sg,mclause:forall,sem:λx.⟦high(x)⟧])
i([symbol:[remain],num:sg])
[remains]
vp([symbol:[remain],mclause:forall,sem:λx.⟦high(x)⟧])
v([symbol:[remain],mclause:forall,sem:λx.⟦high(x)⟧])
iv_adj([symbol:[remain],mclause:forall,sem:λpλx.⟦p(x)⟧])
[ ]
adj([sem:λx′.high(x′)])
[high]

Figure 4.5.15: The structure of sentences (162) and (163), respectively.

when the `iv` category (which stands for Intransitive Verb) combines with the `adj` category as in the bottom phrase structure, we assign to the `iv` category a different semantic type than the `iv` category without the `adj` category as in the top phrase structure. Therefore, we write the `iv` category that combines with the `adj` category in this form "`iv_adj`". The verb "remains" is a state verb and has a simple aspect. Thus, it is assigned with $\lambda p \lambda x.[\![p(x)]\!]$ where λp takes the string p and replace it by the string of the `adj` category "$\lambda x'.high(x')$" and obtains "$\lambda x.[\![high(x)]\!]$". After that, we compute the last result with the string of the `np0` category "*Awvalid*" to produce "$[\![high(Awvalid)]\!]$". On the other hand, the verb "asserted" is an event verb and has a perfective aspect. Thus, it is assigned with $\lambda x.\langle assert(x)\rangle\top$ where λx takes the string x and replace it by the string of the `np0` category "*Awvalid*" and obtains "$\langle assert(Awvalid)\rangle\top$".

Another observation, in our grammar at Figure 4.5.14, is that when the `adj` category has an additional feature called `type`, it means that the `adj` category requires its own specific quantification. Therefore, we ignore the semantics of the `iv_adj` category to avoid redundancy in assigning two quantifications on the `v` category. For a very similar reason, we also ignore the value in the `mclause` feature. Let us illustrate this by an example as follows:

(164) Awvalid must remain stable.

We assign its meaning based on the phrase structure given in Figure 4.5.16. As shown in Figure 4.5.16, we ignore the `sem`'s value of the `iv_adj` category since it combines with the `adj` category that has the `type` feature with an `exists` tag. The adjective "stable" has a special treatment in TPL^+ as explained in Section 4.4. Thus, our grammar treats it differently than any other adjective. Note that the string of the adjective "stable" is $\langle e^{\Leftarrow}\rangle\top$ which is considered to be an event type. Therefore, the adjective "stable" has two lexical rules similar to the lexical rules of the verbs "asserted" and "go". The `type` feature of the `adj` category can be bounded by either `exists` or `init_exists` tag. Again, there is not ambiguity that can be risen from these rules for the similar reasons of the rules in the `iv` and `iv_adj` categories.

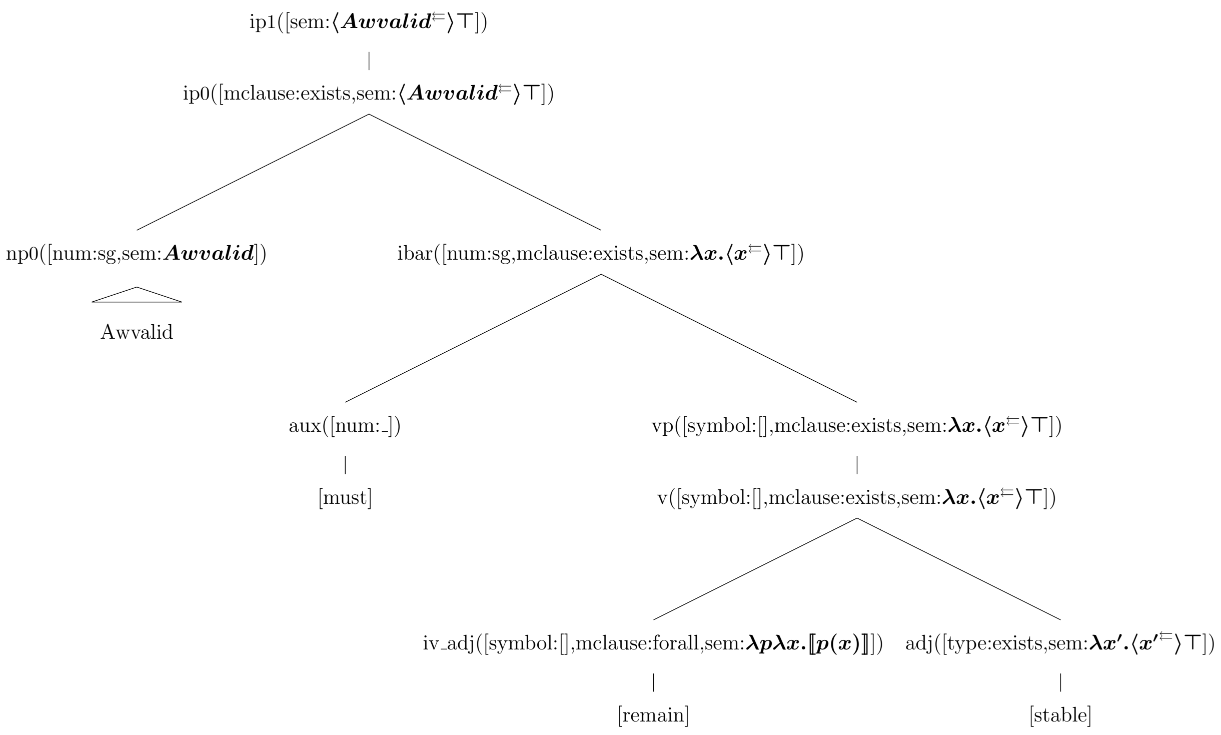

ip1([sem:⟨**Awvalid**⇐⟩⊤])
ip0([mclause:exists,sem:⟨**Awvalid**⇐⟩⊤])
np0([num:sg,sem:**Awvalid**])
Awvalid
ibar([num:sg,mclause:exists,sem:**λx.**⟨**x**⇐⟩⊤])
aux([num:_])
[must]
vp([symbol:[],mclause:exists,sem:**λx.**⟨**x**⇐⟩⊤])
v([symbol:[],mclause:exists,sem:**λx.**⟨**x**⇐⟩⊤])
iv_adj([symbol:[],mclause:forall,sem:**λpλx.**⟦**p(x)**⟧])
[remain]
adj([type:exists,sem:**λx′.**⟨**x′**⇐⟩⊤])
[stable]

Figure 4.5.16: The phrase structure of (164).

Now, let us show how we compute the meaning of a sentence including a negation via our grammar rules by giving the following example:

(165) Awvalid is not asserted.

We assign its meaning using the rules in Figure 4.5.14, as shown in Figure 4.5.17. In Figure 4.5.17, note that combining the `neg` category with the `vp` category affects the quantification type that is stored in the `mclause` feature, where the `exists` quantifier becomes the `forall` quantifier in the `negP` category. We use the predicate `quantifier_modification` in the grammar rules in Figure 4.5.14 to modify the quantifier type based on the additional rules in Table 4.5.2.

M ∈ vp	→	NewM ∈ ibar
exists	→	forall
init_exists	→	forall
forall	→	exists

Table 4.5.2: Quantifier modification.

Based on the rules in the above table, whenever the `neg` category combines with the `vp` category, the variable `NewM` of the `ibar` category will have an quantifier that is opposite the one stored in `M` of the `vp` category. Modifying the quantification type in this level is necessary to provide the proper tag for higher categories. For example, when the `ip0` category combines with the `tpp` category, their tags must match up, otherwise the grammar will reject the sentence as explained earlier with sentences (152) and (153).

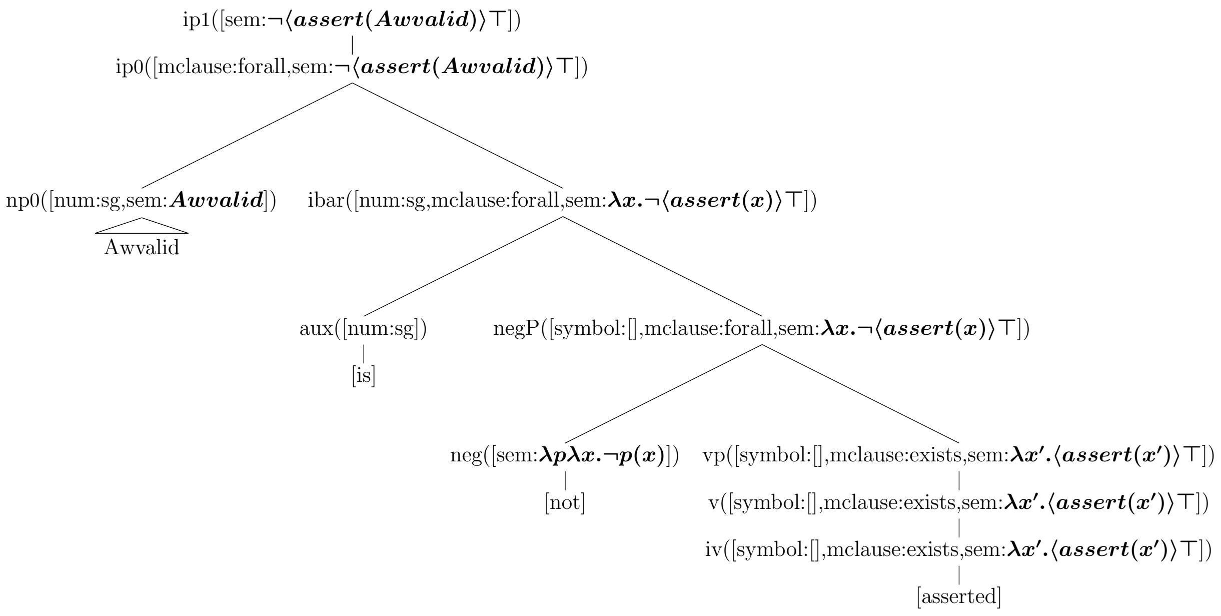

ip1([sem:$\neg\langle \boldsymbol{assert(Awvalid)}\rangle\top$])
ip0([mclause:forall,sem:$\neg\langle \boldsymbol{assert(Awvalid)}\rangle\top$])
np0([num:sg,sem:$\boldsymbol{Awvalid}$])
Awvalid
ibar([num:sg,mclause:forall,sem:$\boldsymbol{\lambda x.}\neg\langle \boldsymbol{assert(x)}\rangle\top$])
aux([num:sg])
[is]
negP([symbol:[],mclause:forall,sem:$\boldsymbol{\lambda x.}\neg\langle \boldsymbol{assert(x)}\rangle\top$])
neg([sem:$\boldsymbol{\lambda p \lambda x.\neg p(x)}$])
[not]
vp([symbol:[],mclause:exists,sem:$\boldsymbol{\lambda x'.}\langle \boldsymbol{assert(x')}\rangle\top$])
v([symbol:[],mclause:exists,sem:$\boldsymbol{\lambda x'.}\langle \boldsymbol{assert(x')}\rangle\top$])
iv([symbol:[],mclause:exists,sem:$\boldsymbol{\lambda x'.}\langle \boldsymbol{assert(x')}\rangle\top$])
[asserted]

Figure 4.5.17: The phrase structure of sentence (165).

4.5.3 Temporal Preposition with nominal complements

Now let us show our grammar rules for temporal prepositions with nominal complements. Temporal preposition phrases (which are denoted by the `tpp` category) combine with the `ip0` category as shown in the grammar rules in Figure 4.5.18. These rules were previously presented in Figure 4.5.4, except the last two rules of the `tpps` category which is used in our grammar rules to allow up to three temporal preposition phrases appear in one sentence.

Again, the variable `M` ranges over `forall`, `exists` and `init_exists`, which is used for the following reason: some temporal prepositions require quantification restrictions over

```
ip2([sem:TPP@IP])-->  ip1([sem:IP]), tpps([sem:TPP]).
ip2([sem:TPP@IP])-->  tpps([sem:TPP]),ip1([sem:IP]).
ip1([sem:TPP@IP])-->  ip0([mclause:M,sem:IP]),
                      tpp([mclause:M,sem:TPP]).
ip1([sem:TPP@IP])-->  tpp([mclause:M,sem:TPP]),
                      ip0([mclause:M,sem:IP]).
tpps([sem:TPP])-->    tpp([mclause:_,sem:TPP]).
tpps([sem:TPP2@TPP1]) -->  tpp([mclause:_,sem:TPP1]),
                           tpp([mclause:_,sem:TPP2]).
```

Figure 4.5.18: Grammar rules for inflectional phrases that consist of temporal preposition phrases

their main clauses as well as their own complements such as *until*, *by*, and *for*. Thus, we write the grammar rule for a temporal preposition with nominal complements as follows:

```
tpp([mclause:M,sem:TP@NP]) -->
     tpn([qclause:Q,mclause:M,sem:TP]), tnp0([type:Q,num:_,sem:NP]).
```

The temporal prepositions with nominal complements belongs to the `tpn` category. Moreover, the variable `Q`, ranging over `forall`, `exists`, `init_exists` and `def`, is used to restrict the complements of the temporal prepositions. Accordingly, we write the closed-class lexicon for temporal prepositions as given in Figure 4.5.19.

As shown in our grammar rules at Figure 4.5.19, λp takes the string p in the string of the `tpn` category and substitute it with the string of temporal noun phrase (`tnp0`) and write one of the modal operators $\{<, <+, >, =\}$ as a subscript of the replaced string. Then λq takes the string q in the string of the `tpn` category and substitute it with the string of the `ip0` category where it is considered the second argument of any of those mentioned operators. Note that λn, on the other hand, takes the string n in $\lambda p.n_{*}p$ and substitute it with the string of temporal noun phrase (`tnp0`) that has a number that restricted by the form $|\ \rangle$ (such as $|3\rangle$). Then we write the modal operator $*$ as a subscript of the replaced string.

In our grammar rules at Figure 4.5.19, we restrict the complements and the main clauses of some temporal prepositions. We follow the same approach in Pratt and Francez's (2001) for restricting some temporal prepositions based on the common use in English. However,

```
tpn([qclause:forall,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [within].
tpn([qclause:def,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [within].
tpn([qclause:init_exists,mclause:exists,sem:$\lambda n\lambda p.n_{*}p$])--> [within].
tpn([qclause:forall,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [at].
tpn([qclause:def,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [at].
tpn([qclause:init_exists,mclause:exists,sem:$\lambda n\lambda p.n_{*}p$])--> [at].
tpn([qclause:forall,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [at].
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [at].
tpn([qclause:init_exists,mclause:forall,sem:$\lambda n\lambda p.n_{*}p$])--> [at].
tpn([qclause:forall,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [during].
tpn([qclause:def,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [during].
tpn([qclause:init_exists,mclause:exists,sem:$\lambda n\lambda p.n_{*}p$])--> [during].
tpn([qclause:forall,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [during].
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [during].
tpn([qclause:init_exists,mclause:forall,sem:$\lambda n\lambda p.n_{*}p$])--> [during].
tpn([qclause:forall,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [on].
tpn([qclause:def,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [on].
tpn([qclause:init_exists,mclause:exists,sem:$\lambda n\lambda p.n_{*}p$])--> [on].
tpn([qclause:forall,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [on].
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [on].
tpn([qclause:init_exists,mclause:forall,sem:$\lambda n\lambda p.n_{*}p$])--> [on].
tpn([qclause:forall,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [for].
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [for].
tpn([qclause:init_exists,mclause:forall,sem:$\lambda n\lambda p.n_{*}p$])--> [for].
tpn([qclause:forall,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [in].
tpn([qclause:def,mclause:exists,sem:$\lambda p\lambda q.p_{=}q$])--> [in].
tpn([qclause:init_exists,mclause:exists,sem:$\lambda n\lambda p.n_{*}p$])--> [in].
tpn([qclause:forall,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [throughout].
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{=}q$])--> [throughout].
tpn([qclause:init_exists,mclause:forall,sem:$\lambda n\lambda p.n_{*}p$])--> [throughout].
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{<}q$])--> [until].
tpn([qclause:def,mclause:exists,sem:$\lambda p\lambda q.p_{<}q$])--> [by].
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{>}q$])--> [since].
tpn([qclause:def,mclause:exists,sem:$\lambda p\lambda q.p_{<}q$])--> [before].
tpn([qclause:forall,mclause:init_exists,sem:$\lambda p\lambda q.p_{<}q$])--> [before].
tpn([qclause:def,mclause:exists,sem:$\lambda p\lambda q.p_{>}q$])--> [after].
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{>}q$])--> [after].
tpn([qclause:forall,mclause:init_exists,sem:$\lambda p\lambda q.p_{>}q$])--> [after].
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{<+}q$])--> [until,after].
```

Figure 4.5.19: Closed-class lexicon for temporal prepositions with nominal complements.

there are other temporal prepositions are not discussed in Pratt and Francez (2001). Thus, we assign to them the most suitable quantification based on the most common use of those temporal prepositions within the natural language protocol specifications.

Note that, in our grammar rules in Figure 4.5.19, each temporal preposition indicating the *during* relation (such as *within*, *for*, *in*, etc) allows its complement to be restricted with `def`, `forall` and `init_exists` quantifiers. The temporal prepositions *at*, *during* and *on* have six rules because they allow their main clauses to be restricted with `exists` and `forall` quantifiers. On the other hand, the temporal prepositions *within*, *for*, *in* and *throughout* have only three rules since they permit their main clauses to be restricted only with one type of quantifier. Another note is that when the complement of a temporal preposition is restricted with `init_exists` quantifier, the meaning of that temporal preposition will be different than the other rules. Finally, no temporal preposition restricts its main clause with existential and initial existential quantifications together. Therefore, we avoid any ambiguity that might occur from the predicate phrase rules in Figure 4.5.14, where any verb can have either existential quantification or initial existential quantification. This leads us to have two different meanings and that is not acceptable by our TempCNL.

Let us explain the grammar rules in Figure 4.5.19 by showing a few examples where those grammar rules work. Consider the following examples

(166) during every reset

(167) until the response phase

(168) after the last transaction.

The phrase structures of the above examples are given in Figures 4.5.20, 4.5.21 and 4.5.22, respectively. In the phrase structure of example (166), the preposition *during* has universal quantification on its complement while in the phrase structures of examples (167) and (168), the prepositions *until* and *after* have a definite quantification on their complements.

Moreover, the prepositions *during* and *after* have the variable `M` on their main clauses, which means it can be either restricted with existential or universal quantifier while the preposition *until* is restricted its main clause with universal quantification.

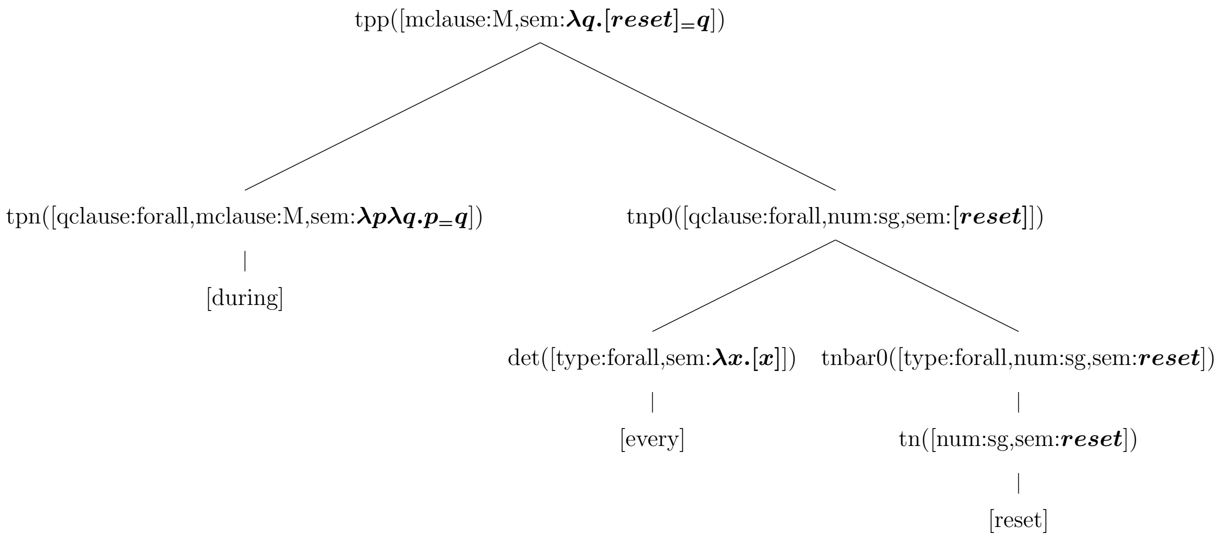

tpp([mclause:M,sem:$\lambda q.[reset]_{=}q$])
tpn([qclause:forall,mclause:M,sem:$\lambda p\lambda q.p_{=}q$])
[during]
tnp0([qclause:forall,num:sg,sem:[$reset$]])
det([type:forall,sem:$\lambda x.[x]$])
[every]
tnbar0([type:forall,num:sg,sem:$reset$])
tn([num:sg,sem:$reset$])
[reset]

Figure 4.5.20: The structure of phrase (166).

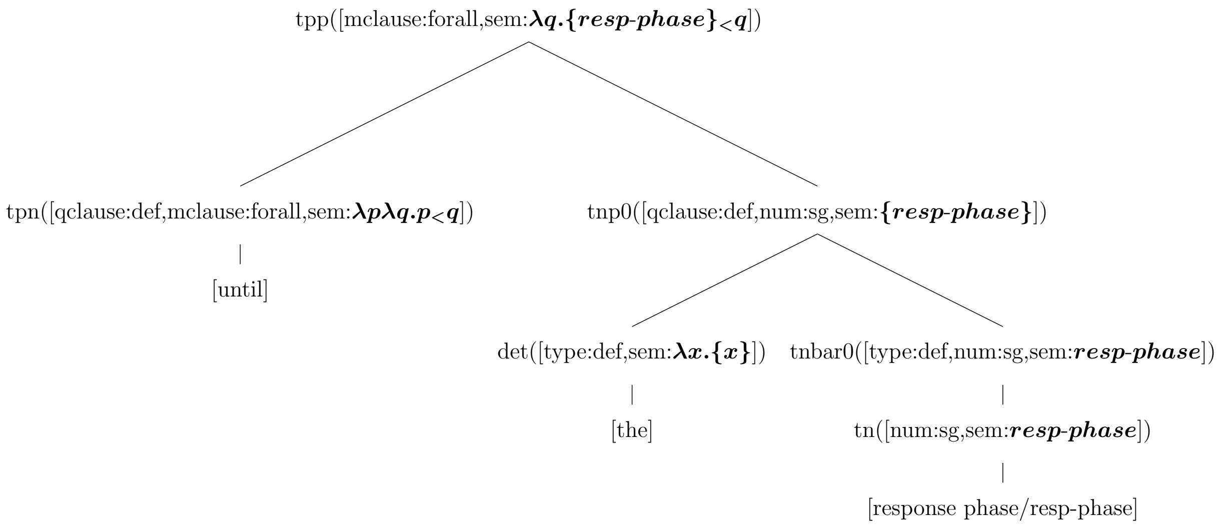

tpp([mclause:forall,sem:$\lambda q.\{resp\text{-}phase\}_{<}q$])
tpn([qclause:def,mclause:forall,sem:$\lambda p\lambda q.p_{<}q$])
[until]
tnp0([qclause:def,num:sg,sem:$\{resp\text{-}phase\}$])
det([type:def,sem:$\lambda x.\{x\}$])
[the]
tnbar0([type:def,num:sg,sem:$resp\text{-}phase$])
tn([num:sg,sem:$resp\text{-}phase$])
[response phase/resp-phase]

Figure 4.5.21: The structure of phrase (167).

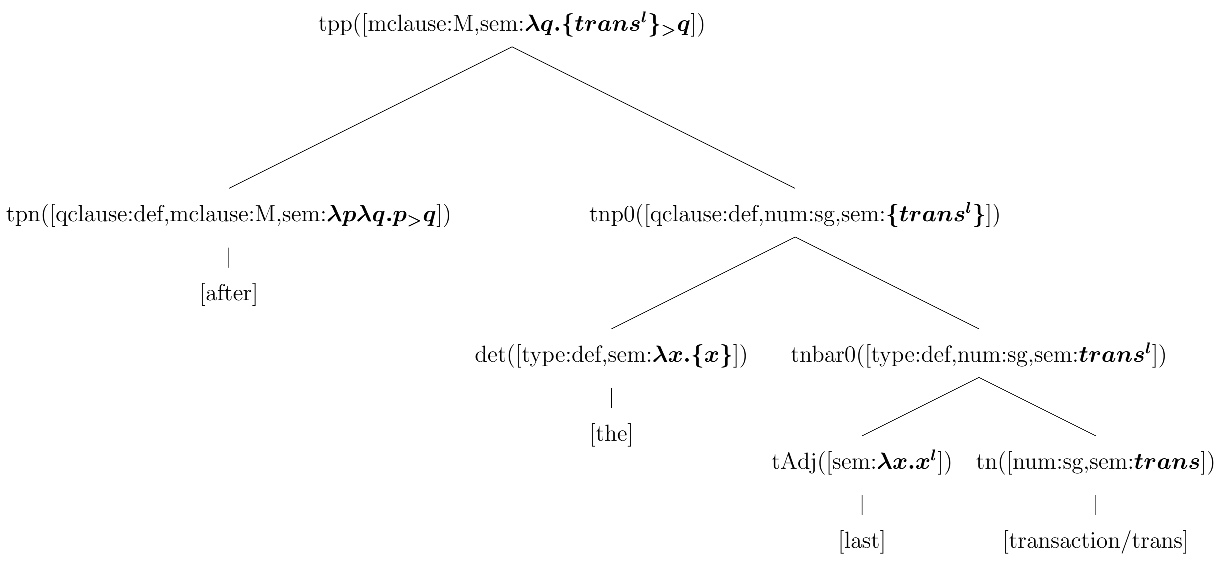

tpp([mclause:M,sem:$\boldsymbol{\lambda q.\{trans^l\}_> q}$])
tpn([qclause:def,mclause:M,sem:$\boldsymbol{\lambda p \lambda q.p_> q}$])
[after]
tnp0([qclause:def,num:sg,sem:$\boldsymbol{\{trans^l\}}$])
det([type:def,sem:$\boldsymbol{\lambda x.\{x\}}$])
[the]
tnbar0([type:def,num:sg,sem:$\boldsymbol{trans^l}$])
tAdj([sem:$\boldsymbol{\lambda x.x^l}$])
[last]
tn([num:sg,sem:$\boldsymbol{trans}$])
[transaction/trans]

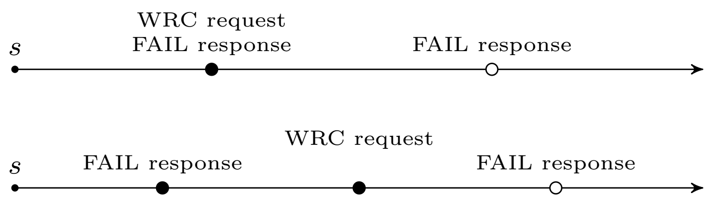
Figure 4.5.22: The structure of phrase for (168).

Note that in our grammar rules in Figure 4.5.19, no temporal prepositions can take indefinite articles (such as *a*, *an* and *some*) in their complements, and here is why. Consider the following sentences:

(169) FAIL response must occur on a WRC request.

(170) FAIL response must occur before a WRC request.

Both sentences are acceptable in general English. "a WRC request" asserts that there is one instance over which WRC request is true and the occurrence of "FAIL response" must be evaluated during that instance as in (169) or before it as in (170). However, these sentences are considered odd in natural language protocol requirements because what if "FAIL response" occurs more than one and "WRC request" occurs once during the evaluation? This can be shown as follows:



where the first "FAIL response", in both diagrams, is satisfied the conditions that are given in (169) and (170) with respect to the event “WRC request”. However, the second “FAIL response” occurs freely without any constrain, and that contradicts the conditions that are given in (169) and (170). The purpose of using these requirements is to apply them for all “FAIL responses” rather than one “FAIL response”. Thus, we must constrain all “FAIL response” to be evaluated on every time over which “WRC request” occurs or on a unique time over which “WRC request” during the evaluation. Thus, we choose not to accept sentences having indefinite articles in the complements of temporal prepositions.

Another feature of our grammar rules in Figure 4.5.19 is that any temporal preposition having the modal operator $*$ in its meaning can combine only with the `tnp0` category having `init_exists` tag in the `qclause` feature. This is because a cardinal number with a temporal noun is the only type that can have the modal operator $*$ as explained after the rules in Figure 4.5.9 in Section 4.5.2.1. On the other hand, if those temporal prepositions do not have a `tnp0` category that consists of a cardinal number and a temporal noun, then we have different interpretations for them. Consider the following examples:

(171) Awid goes high in 3 cycles

(172) Awid goes high in every reset

where sentence (171) asserts that, within the temporal context *I*, the initial interval of *I* over which 3 cycles occur includes an interval over which *Awid* becomes high; while sentence (172) asserts that, within the temporal context *I*, every interval over which reset occurs includes an interval over which *Awid* becomes high. Thus, the semantics of the preposition *in* in the above examples differ. Hence, we produce their phrase structures differently using our grammar rules as shown in Figure 4.5.23.

Let us give another justification for these restrictions. Some prepositions may come with more than one quantification, such as the prepositions *before* and *after* such as the following examples:

(173) Awid goes high (before/after) the response phase

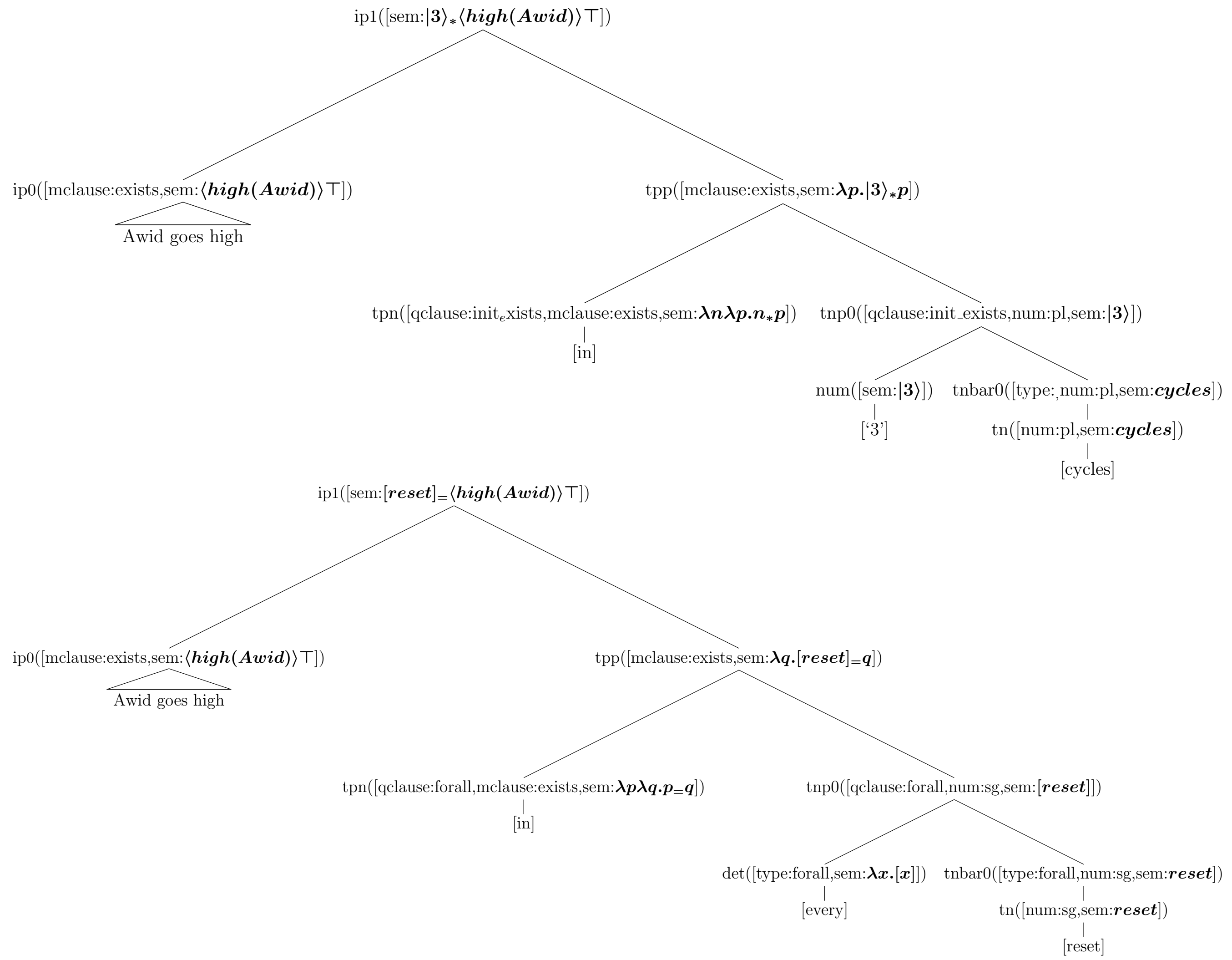

ip1([sem:|3⟩∗⟨high(Awid)⟩⊤])
ip0([mclause:exists,sem:⟨high(Awid)⟩⊤])
Awid goes high
tpp([mclause:exists,sem:λp.|3⟩∗p])
tpn([qclause:init_exists,mclause:exists,sem:λnλp.n∗p])
[in]
tnp0([qclause:init_exists,num:pl,sem:|3⟩])
num([sem:|3⟩])
['3']
tnbar0([type:_,num:pl,sem:cycles])
tn([num:pl,sem:cycles])
[cycles]
ip1([sem:[reset]=⟨high(Awid)⟩⊤])
ip0([mclause:exists,sem:⟨high(Awid)⟩⊤])
Awid goes high
tpp([mclause:exists,sem:λq.[reset]=q])
tpn([qclause:forall,mclause:exists,sem:λpλq.p=q])
[in]
tnp0([qclause:forall,num:sg,sem:[reset]])
det([type:forall,sem:λx.[x]])
[every]
tnbar0([type:forall,num:sg,sem:reset])
tn([num:sg,sem:reset])
[reset]

Figure 4.5.23: The phrase structures of sentences (171) and (172).

(174) Awid goes high (before/after) every response phase.

In sentence (173), the terms *before* and *after* are interpreted in the sense of *some time before* and *some time after*, respectively. However, in sentence (174), the terms *before* and *after* are interpreted in the sense of *shortly-before* and *shortly-after*, respectively. Thus, we write grammar rules for *shortly-before* and *shortly-after* interpretations as follows:

```
tpn([qclause:forall,mclause:init_exists,sem:λpλq.p_{<}q])--> [before].
tpn([qclause:forall,mclause:init_exists,sem:λpλq.p_{>}q])--> [after].
```

where the `init_exists` tag enforces the semantics of the main clauses (which are represented here by the string q) to be *shortly-before* or *shortly-after*.

Let us produce the phrase structure for the preposition *after* in sentence (174) using the above rules, as shown in Figure 4.5.24.

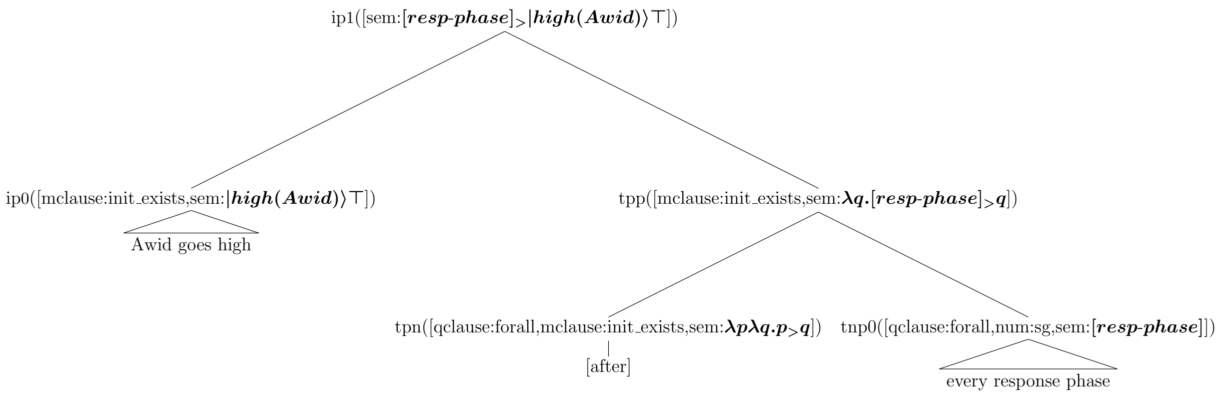

ip1([sem:[resp-phase]$_>$|high(Awid)⟩⊤])
ip0([mclause:init_exists,sem:|high(Awid)⟩⊤])
Awid goes high
tpp([mclause:init_exists,sem:λq.[resp-phase]$_>$q])
tpn([qclause:forall,mclause:init_exists,sem:λpλq.p$_>$q])
[after]
tnp0([qclause:forall,num:sg,sem:[resp-phase]])
every response phase

Figure 4.5.24: The phrase structure for the preposition *after* in sentence (174).

Our grammar produces $|high(awid)\rangle\top$ for the sentence "Awid goes high" since the preposition *after* requires its main clause to be restricted with initial existential quantifier $|\ \rangle\top$. We make this restriction by adding `init_exists` tag to the rule of the temporal preposition *after*, where the complement of the temporal preposition *after* must be universaly quantified.

4.5.4 Temporal Prepositions with Clausal complements

Now let us turn to our grammar rules for temporal prepositions with clausal complements, as denoted by the `tps` category, as shown in Figure 4.5.25.

```
tpp([mclause:M,sem:TP@IP]) --> tps([qclause:_,mclause:M,sem:TP]),
                               removeQ([mclause:_,sem:IP]).
removeQ([mclause:_,sem:e]) --> ip0([mclause:_,sem:⟨e⟩⊤]).
removeQ([mclause:_,sem:e]) --> ip0([mclause:_,sem:⟦e⟧]).
```

Figure 4.5.25: The grammar rules for temporal prepositions with clausal complements.

Each temporal preposition with a clausal complement has its own quantification restriction in our grammar. Thus, we must remove any quantification restriction (either existential or universal) that rises from the `ip0` category. This can be done by using an additional category called `removeQ` which stands for removing quantifiers. This category takes only the event e and drops the bracket-pair $\langle\ \rangle\top$ or the bracket-pair $[\![\ ]\!]$ that occurs in the `ip0` category as shown in Figure 4.5.25.

(175) Awid must remain high until Awready goes high.

where it is perfectly natural to have in *until*'s complement a sentence that is restricted with existential quantification; however when we interpret it into the TPL^+ logic, we prefer *until* with definite quantification as shown in the following TPL^+ formulas:

(176) $?\ \langle high(Awready)\rangle_{<}[\![high(Awid)]\!]$,

(177) $\{high(Awready)\}_{<}[\![high(Awid)]\!]$.

Formula (176) is slightly odd since the modal operator $<$ (which stands for the relation *before*) does not accept existential quantification on its left argument, unlike formula (177) where definite quantification on the left of the modal operator $<$ is more natural. Note that the topic of enforcing *until*'s complement to be definite quantified is explained before in examples (83) in Section 3.3.1.

Now, we write the closed-class lexicon for temporal preposition with clausal complements as given in Figure 4.5.26.

```
tps([qclause:forall,mclause:exists,sem:$\lambda p\lambda q.[p]_{=}q$])--> [when].
tps([qclause:forall,mclause:forall,sem:$\lambda p\lambda q.[p]_{=}q$])--> [when].
tps([qclause:forall,mclause:exists,sem:$\lambda p\lambda q.[p]_{=}q$])--> [whilst].
tps([qclause:forall,mclause:forall,sem:$\lambda p\lambda q.[p]_{=}q$])--> [whilst].
tps([qclause:forall,mclause:exists,sem:$\lambda p\lambda q.[p]_{=}q$])--> [while].
tps([qclause:forall,mclause:forall,sem:$\lambda p\lambda q.[p]_{=}q$])--> [while].
tps([qclause:forall,mclause:exists,sem:$\lambda p\lambda q.[p]_{=}q$])--> [whenever].
tps([qclause:forall,mclause:forall,sem:$\lambda p\lambda q.[p]_{=}q$])--> [whenever].
tps([qclause:def,mclause:forall,sem:$\lambda p\lambda q.\{p\}_{<}q$])--> [until].
tps([qclause:def,mclause:exists,sem:$\lambda p\lambda q.\{p\}_{<}q$])--> [before].
tps([qclause:def,mclause:exists,sem:$\lambda p\lambda q.\{p\}_{<}q$])--> [by,the,time].
tps([qclause:def,mclause:exists,sem:$\lambda p\lambda q.\{p\}_{>}q$])--> [after].
tps([qclause:def,mclause:forall,sem:$\lambda p\lambda q.\{p\}_{>}q$])--> [after].
tps([qclause:forall,mclause:init_exists,sem:$\lambda p\lambda q.[p]_{>}q$])--> [once].
tps([qclause:def,mclause:forall,sem:$\lambda p\lambda q.\{p\}_{>}q$])--> [since].
tps([qclause:def,mclause:forall,sem:$\lambda p\lambda q.\{p\}_{<+}q$])--> [until,after].
```

Figure 4.5.26: Closed-class lexicon for temporal prepositions with clausal complements.

In our grammar rules at Figure 4.5.26, each temporal preposition indicating the *during* relation (such as *when*, *while*, etc) allows its main clause to be restricted with `exists` and `forall` quantifiers. Therefore, each of them has two rules. Moreover, the temporal preposition *after* also has two rules since it allows its main clause to be restricted with `exists` and `forall` quantifiers. Finally, the only temporal preposition that allows its main clause to be restricted with `init_exists` quantifier is the temporal preposition *once* because it locates the event in its main clause immediately after the event in its complement finishes.

Let us begin explaining the above rules with the following example:

(178) whenever Awvalid goes low

(179) until Awready goes high

The phrase structures of (178) and (179) are shown in Figure 4.5.27.

The phrase structure (a) shows that *whenever* is only associated with universal quantification. Thus, we must remove the existential quantification from $\langle low(Awvalid)\rangle\top$

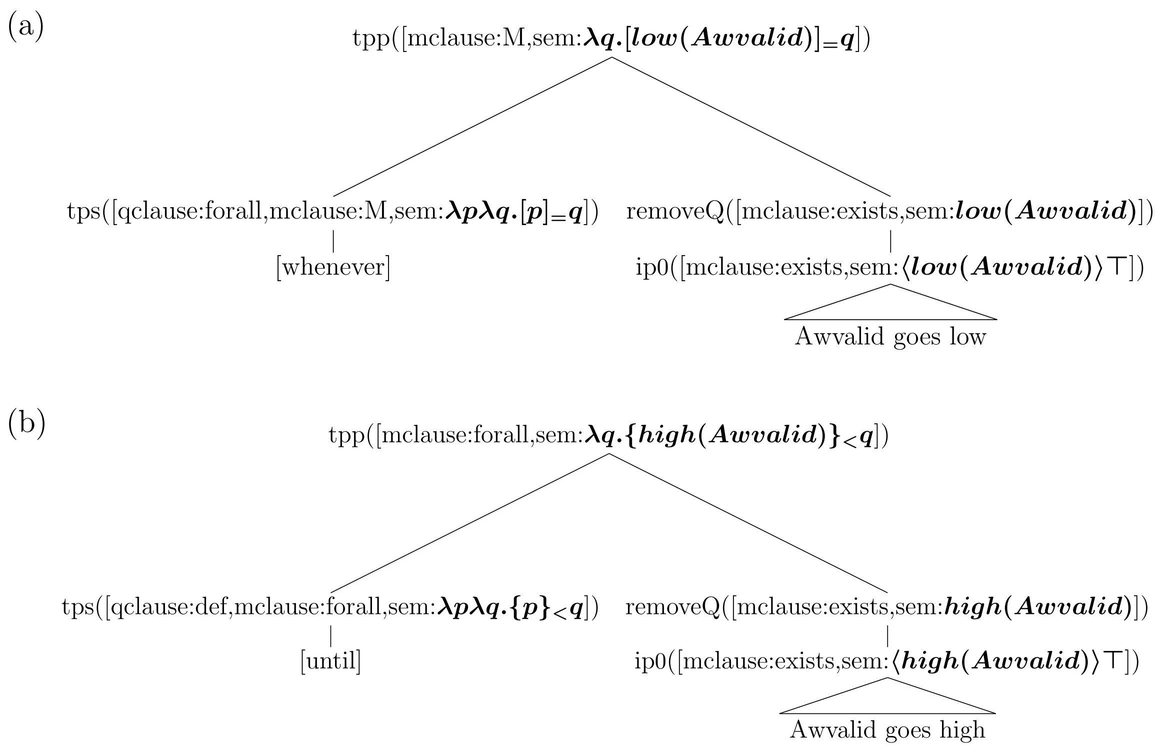

(a)
tpp([mclause:M,sem:$\boldsymbol{\lambda q.[low(Awvalid)]_{=}q}$])
tps([qclause:forall,mclause:M,sem:$\boldsymbol{\lambda p \lambda q.[p]_{=}q}$])
[whenever]
removeQ([mclause:exists,sem:$\boldsymbol{low(Awvalid)}$])
ip0([mclause:exists,sem:$\langle \boldsymbol{low(Awvalid)} \rangle \top$])
Awvalid goes low
(b)
tpp([mclause:forall,sem:$\boldsymbol{\lambda q.\{high(Awvalid)\}_{<}q}$])
tps([qclause:def,mclause:forall,sem:$\boldsymbol{\lambda p \lambda q.\{p\}_{<}q}$])
[until]
removeQ([mclause:exists,sem:$\boldsymbol{high(Awvalid)}$])
ip0([mclause:exists,sem:$\langle \boldsymbol{high(Awvalid)} \rangle \top$])
Awvalid goes high

Figure 4.5.27: The phrase structures of (178) and (179).

by moving only the string $low(Awvalid)$ to the `removeQ` category. Then, λp takes the string p in $\lambda p \lambda q.[p]_{=}q$ and substitute it with the string $low(Awvalid)$. Now, $low(Awvalid)$ becomes restricted by universal quantification rather than existential one. On the other hand, the phrase structure (b) shows that *until* is only associated with definite quantification. Thus, we must remove the existential quantification from $\langle high(Awready) \rangle \top$ by moving only the string $high(Awvalid)$ to the `removeQ` category. Then, λp takes the string p in $\lambda p \lambda q.\{p\}_{<}q$ and substitute it with the string $high(Awready)$. Now, $high(Awready)$ becomes restricted by definite quantification rather than existential one. These phrase structures show that applying these restrictions on the semantics of temporal prepositions help to produce correct TPL^{+} formulas that correspond to the truth-condition of English sentences. Note that, in the phrase structure (a), the preposition *whenever* has the variable `M` on its `mclause` feature, which means the variable `M` can be bound by either existential or universal quantifier based on its lexicon rules in Figure 4.5.26.

Let us show how to extract a TPL^{+} formula from a sentence having two temporal prepositions based on our grammar rules. Consider the following sentence:

(180) After Awvalid becomes low, Awid remains asserted until Awready goes high.

We assign to sentence (180) the phrase structure shown in Figure 4.5.28. As shown in Figure 4.5.28, combining the `tpps` category with the `ip1` category that has another `tpp` category works perfectly using our grammar rules in Figure 4.5.18. Moreover, the temporal preposition *after* has the variable `M` on its `mclause` feature; however this variable will be ignored since it has no effect during the process of extracting the TPL^+ formula.

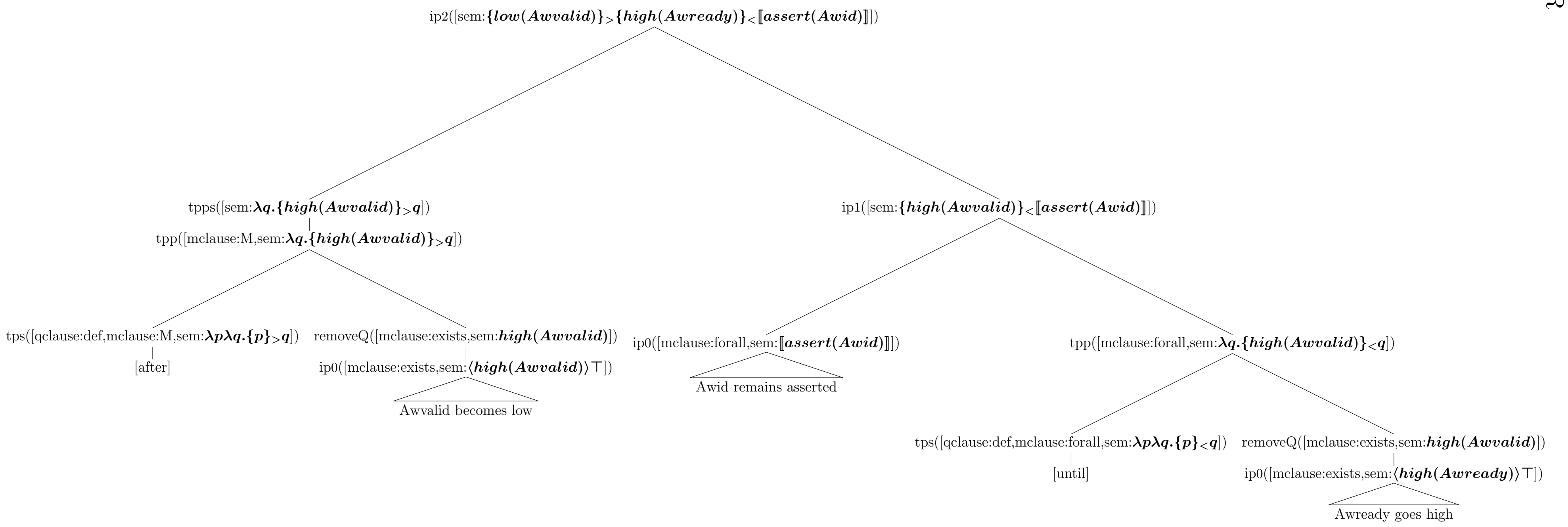

ip2([sem:{low(Awvalid)}>{high(Awready)}<⟦assert(Awid)⟧])
tpps([sem:λq.{high(Awvalid)}>q])
tpp([mclause:M,sem:λq.{high(Awvalid)}>q])
tps([qclause:def,mclause:M,sem:λpλq.{p}>q])
[after]
removeQ([mclause:exists,sem:high(Awvalid)])
ip0([mclause:exists,sem:⟨high(Awvalid)⟩⊤])
Awvalid becomes low
ip1([sem:{high(Awvalid)}<⟦assert(Awid)⟧])
ip0([mclause:forall,sem:⟦assert(Awid)⟧])
Awid remains asserted
tpp([mclause:forall,sem:λq.{high(Awvalid)}<q])
tps([qclause:def,mclause:forall,sem:λpλq.{p}<q])
[until]
removeQ([mclause:exists,sem:high(Awvalid)])
ip0([mclause:exists,sem:⟨high(Awready)⟩⊤])
Awready goes high

Figure 4.5.28: The phrase structure of sentence (180).

4.5.5 Coordination

In this section, we extend our grammar rules for a variety of forms of coordination. Coordination is possible for sentences, noun phrases and temporal nominal phrases. For each phrase, let us show how our grammar rules work with coordination.

Let sentence coordination and its associated rules be defined by the phrase structure rules:

```
ip2([sem:(B@A)@C]) --> ip1([sem:A]), ipcoord([sem:B]), ip1([sem:C]).
ipcoord([sem:λpλq.p ∧ q]) --> [and].
ipcoord([sem:λpλq.p ∨ q]) --> [or].
```

We can apply the above phrase structure rules to extract the semantics in TPL^+ for a sentence like "Rvalid is asserted and Awready is low " as shown in Figure 4.5.29.

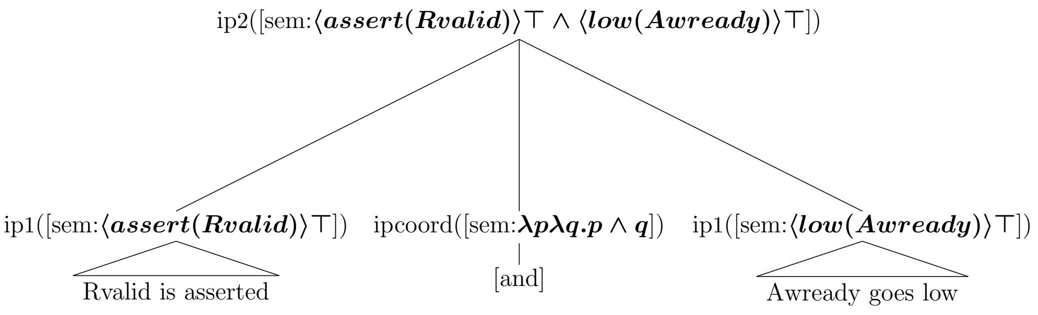

ip2([sem:⟨assert(Rvalid)⟩⊤ ∧ ⟨low(Awready)⟩⊤])
ip1([sem:⟨assert(Rvalid)⟩⊤])
ipcoord([sem:λpλq.p ∧ q])
ip1([sem:⟨low(Awready)⟩⊤])
Rvalid is asserted
[and]
Awready goes low

Figure 4.5.29: The phrase structure of a sentence coordination.

We now turn to noun phrase coordination and its associated rules. We write the phrase structure rules for them as follows:

```
ip0([mclause:M,sem:NP@IBar])-->  np1([num:Num,sem:NP]),
                                 ibar([num:Num,mclause:M,sem:IBar]).
np1([num:Num,sem:(B@A)@C]) -->
        np0([num:Num,sem:A]),npcoord([sem:B]), np0([num:Num,sem:C]).
npcoord([sem:λpλqλs.s(p) ∧ s(q)]) --> [and].
npcoord([sem:λpλqλs.s(p) ∨ s(q)]) --> [or].
```

Note the `ip0` category consists of the `np1` category, followed by the `ibar` category. Thus, we apply the `sem`'s value of the `ibar` category to the `sem`'s value of the `np1` category rather than the opposite direction since the `np1` has the `npcoord` category which requires us to use the application operator in this direction to extract the meaning of the given sentences that include noun phrase coordination. For example, we can extract the TPL^+ formula for a sentence like "Rvalid or Awready becomes low" using the above phrase structure rules as shown in Figure 4.5.30.

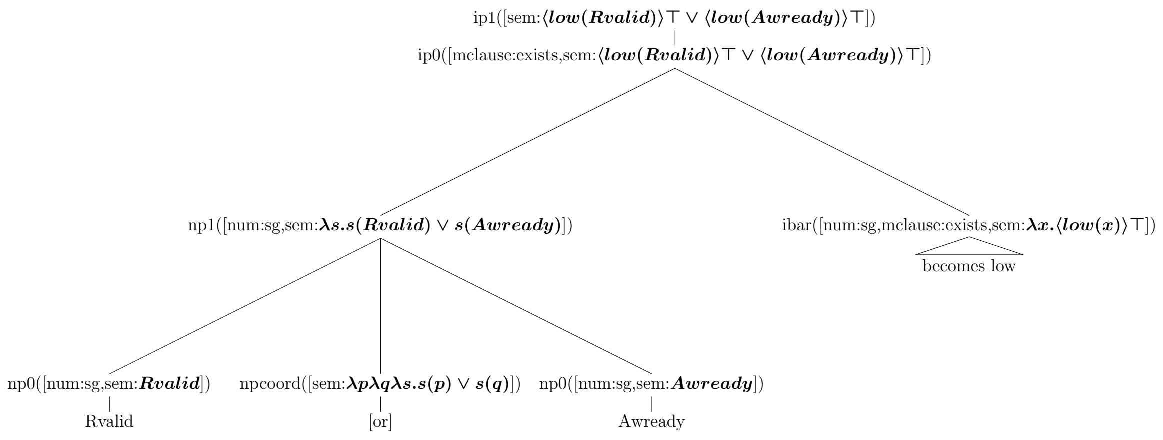

ip1([sem:⟨low(Rvalid)⟩⊤ ∨ ⟨low(Awready)⟩⊤])
ip0([mclause:exists,sem:⟨low(Rvalid)⟩⊤ ∨ ⟨low(Awready)⟩⊤])
np1([num:sg,sem:λs.s(Rvalid) ∨ s(Awready)])
ibar([num:sg,mclause:exists,sem:λx.⟨low(x)⟩⊤])
becomes low
np0([num:sg,sem:Rvalid])
Rvalid
npcoord([sem:λpλqλs.s(p) ∨ s(q)])
[or]
np0([num:sg,sem:Awready])
Awready

Figure 4.5.30: The phrase structure of a noun phrase coordination.

The final form of coordination we consider is that of temporal nouns. Let temporal noun coordination and its associated rules be defined by the phrase structure rules:

```
tpp([mclause:M,sem:NP@TP]) -->
    tpn([qclause:Q,mclause:M,sem:TP]), tnp1([type:Q,num:_,sem:NP]).
tnp1([qclause:D,num:Num,sem:TnBar@Det]) -->
    det([type:D,sem:Det]), tnbar1([type:D,num:Num,sem:TnBar]).
tnbar1([type:_,num:Num,sem:(B@A)@C]) -->
    tn([num:Num,sem:A]), tncoord([sem:B]), tn([num:Num,sem:C]).
tncoord([sem:λpλqλzλs.s(z(p)) ∧ s(z(q))]) --> [and].
tncoord([sem:λpλqλzλs.s(z(p)) ∨ s(z(q))]) --> [or].
```

When the `tpp` category consists of the `tpn` category, followed by the `tnp1` category, we apply the `sem`'s value of the `tpn` category to the `sem`'s value of the `tnp1` category. Moreover, when the `tnp1` category consists of the `det` category, followed by the `tnbar1` category, we apply the `sem`'s value of the `det` category to the `sem`'s value of the `tnbar1` category. Combining the `sem`'s values in this order is important to extract the meaning of the given sentences that include temporal noun coordination. For example, we can extract the TPL^+ formula for a sentence like "Rvalid is asserted during every request and acknowledgement" using the above phrase structure rules as shown in Figure 4.6.1.

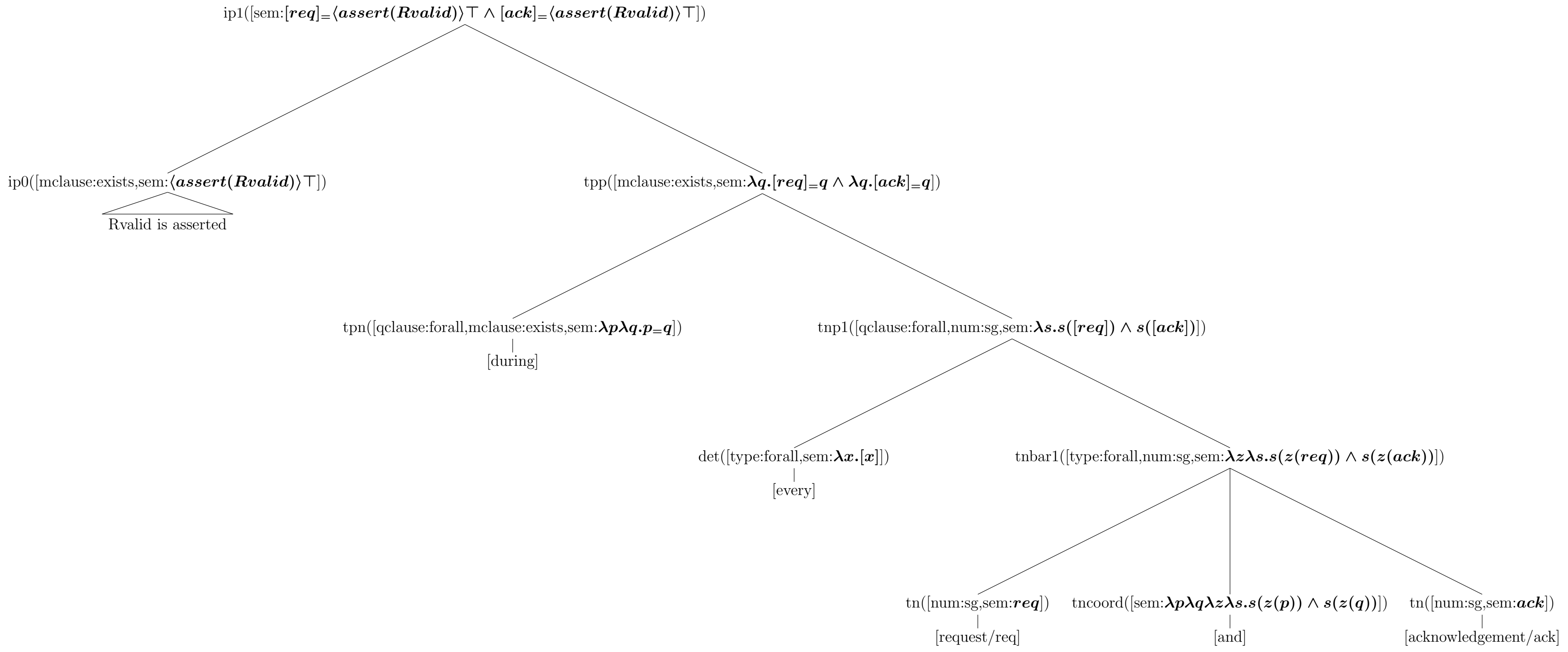

ip1([sem:$[\boldsymbol{req}]_=\langle \boldsymbol{assert(Rvalid)}\rangle\top \wedge [\boldsymbol{ack}]_=\langle \boldsymbol{assert(Rvalid)}\rangle\top$])
ip0([mclause:exists,sem:$\langle \boldsymbol{assert(Rvalid)}\rangle\top$])
Rvalid is asserted
tpp([mclause:exists,sem:$\boldsymbol{\lambda q.[req]_=q} \wedge \boldsymbol{\lambda q.[ack]_=q}$])
tpn([qclause:forall,mclause:exists,sem:$\boldsymbol{\lambda p\lambda q.p_=q}$])
[during]
tnp1([qclause:forall,num:sg,sem:$\boldsymbol{\lambda s.s([req])} \wedge \boldsymbol{s([ack])}$])
det([type:forall,sem:$\boldsymbol{\lambda x.[x]}$])
[every]
tnbar1([type:forall,num:sg,sem:$\boldsymbol{\lambda z\lambda s.s(z(req))} \wedge \boldsymbol{s(z(ack))}$])
tn([num:sg,sem:$\boldsymbol{req}$])
[request/req]
tncoord([sem:$\boldsymbol{\lambda p\lambda q\lambda z\lambda s.s(z(p))} \wedge \boldsymbol{s(z(q))}$])
[and]
tn([num:sg,sem:$\boldsymbol{ack}$])
[acknowledgement/ack]

Figure 4.6.1: The phrase structure of a temporal noun coordination.

4.6 Restricted TPL^+

In this section, we restrict some TPL^+ formulas defined in Section 4.4 to simplify their mapping into *LTL* and *SVA* as described in the next Chapter. The TPL^+ formulas that require to be more restricted as follows: $\{e\}\psi$, $\{e\}_>\psi$, $\{e\}_<\psi$ and $\{e\}_{<+}\psi$. Let us start by motivating these restrictions using some examples.

Consider the following sentences:

(181) Awid remains high throughout the Ack;

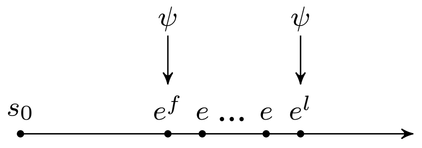
(182) Awid must be high after the Ack;

The corresponding TPL^+ formulas for the above sentences are as follows, respectively:

(183) $\{Ack\}[\![high(Awid)]\!]$

(184) $\{Ack\}_>\langle high(Awid)\rangle\top$.

The TPL^+ formulas (183) and (184) have the same truth conditions as the sentences (181) and (182). However, mapping these kinds of TPL^+ formulas into *LTL* and *SVA* is hard because the definite quantifier does not exist in the semantics of *LTL* and *SVA*. However, if the definite quantifier comes with the *event-relation* category e^f or e^l, then we can express them in *LTL* as we shall see in Section 5.2. The event e (such as *Ack*) might occur several times during the evaluation as shown below:



Therefore, we restrict the event e to be the first e or the last e to avoid evaluating the embedded formula (such as ψ in the previous time line) in the other occurrences of e. For example, if we evaluate ψ in the time line, we must evaluate it only with respect to the first e or the last e and ignore the rest of e being occurred.

Here is how we can make these modifications on such TPL^+ formulas. After they are generated using our grammar rules (discussed in Section 4.5), if an TPL^+ formula occurs in the following forms:$\{e\}\psi$, $\{e\}_>\psi$, $\{e\}_<\psi$ and $\{e\}_{<+}\psi$ then we rewrite them as shown in the following cases:

- $\{e\}\psi \Rightarrow \{e^f\}\psi$
- $\{e\}_>\psi \Rightarrow \{e^l\}_>\psi$
- $\{e\}_<\psi \Rightarrow \{e^f\}_<\psi$
- $\{e\}_{<+}\psi \Rightarrow \{e^f\}_{<+}\psi$.

Now, let us apply these modifications using the above rules on TPL^+ formulas (183) and (184) as follows, respectively

(185) $\{Ack^f\}[\![high(Awid)]\!]$

(186) $\{Ack^l\}_>\langle high(Awid)\rangle\top$

The reason for not applying these modifications during the process of extracting TPL^+ formulas is to avoid the case where sentences already have the adjective first and last. This case will cause some duplications on the *event-relation* category e^f or e^l as shown in (187) and (188).

(187) * $\{e^{ff}\}\psi$

(188) * $\{e^{ll}\}_>\psi$

Thus, applying these modifications after TPL^+ formulas are generated is consider the best solution for this issue. From now on, we rewrite the mentioned forms of TPL^+ as suggested above.

4.7 Rewriting TPL^+ Primitives

In this section, we rewrite TPL^+ primitives to simplify the translation process from TPL^+ to *SVA* as we see in the next Chapter. TPL^+ primitives will be rewritten after they are generated using our grammar rules. We start by introducing TPL^+ primitives that require some simplifications. Then, we show how we rewrite those TPL^+ primitives.

In practice, there are some common TPL^+ primitives that are interpreted in the same way in *SVA* as shown in Table 4.7.1.

Group A		Group B	
assert(ψ)	permit(ψ)	deassert(ψ)	disabled(ψ)
hold(ψ)	valid(ψ)	illegal(ψ)	absent(ψ)
active(ψ)	occur(ψ)	inactive(ψ)	invalid(ψ)
enabled(ψ)	high(ψ)	low(ψ)	

Table 4.7.1: A list of common primitives which share the same meanings in *SVA*.

Indeed, those primitives can have different meanings if they occur in other domains. However, in *SVA*, the primitives in Group A mean that ψ is true while the primitives in Group B mean that ψ is false since they have implicitly negative readings such as $deassert(\psi)$ and $low(\psi)$ as described in Section 2.4.

Let us now show how we can simplify the primitives in Table 4.7.1. Consider the following TPL^+ formulas for representing the meanings of sentences (131) and (139), respectively:

(189) $[cycle]\langle assert(Awid)\rangle\top$

(190) $|3\rangle_*[\![high(Awvalid)]\!]$

where $assert(Awid)$ and $high(Awvalid)$ are unary predicate constants. Our simplification requires us to (i) remove any unary predicates in Table 4.7.1 and (ii) replace it by capitalising its argument to denote the unary predicate and its argument together. For example, we can simplify the TPL^+ formulas (189) and (190) as follows, respectively:

(191) $[CYCLE]\langle AWID\rangle\top$

(192) $|3\rangle_*[\![AWVALID]\!]$

where $CYCLE$ is also rewritten using uppercase letters. Now, $AWID$ and $AWVALID$

are considered as predicates with zero arguments. Moreover, $AWID$ and $AWVALID$ denote that $AWID$ and $AWVALID$ respectively are true at the time in question. However, the primitives in Group B at Table 4.7.1 must be treated differently than the above simplifications since they have implicitly negative reading. For example, consider the following TPL^+ formula for representing the meaning of sentence (180):

(193) $\{low(Awvalid)\}_>\{high(Awready)\}_<[\![assert(Awid)]\!]$,

where $high(Awready)$ and $assert(Awid)$ can be simplified in the similar way of the previous examples. On the other hand, $low(Awvalid)$ must be simplified differently by (i) replacing the unary predicate low by the negation symbol $\neg$ and (ii) capitalising its argument ($Awvalid$) to denote that its argument is restricted by a negation as shown below:

(194) $\{\neg AWVALID\}_>\{AWREADY\}_<[\![AWID]\!]$,

where $\neg AWVALID$ denotes that $AWVALID$ is false at the time in question.

To summarize, any TPL^+ primitive from Table 4.7.1 will be rewritten as shown in Table 4.7.2. These modifications are essential for simplifying the translation process from TPL^+ to SVA as we shall see in the next chapter.

Group	TPL^+		
A	assert(ψ), permit(ψ), hold(ψ), valid(ψ), active(ψ), occur(ψ), enabled(ψ), and high(ψ)	$\Rightarrow$	Ψ
B	de-assert(ψ), disabled(ψ), absent(ψ), illegal(ψ), inactive(ψ), invalid(ψ), and low(ψ)	$\Rightarrow$	$\neg\Psi$

Table 4.7.2: A list of some common TPL^+ primitives and their simplifications.

Beside rewriting these TPL^+ primitives, we also rewrite primitives' arguments, which denoted by ψ in Table 4.7.2, using the capital letters as part of our simplification process. Here we represent writing ψ with the capital letters as Ψ. Form now, we shall rewrite TPL^+ formulas using these simplification rules.

Chapter 5

Generating *LTL* and *SVA* from TPL^+

In this chapter, we consider the problem of translating TPL^+ to *SVA* (a variant of *LTL*). We certainty have reason to believe that such a translation is possible. After all, consideration of the semantics of TPL^+ suggests that TPL^+ can be translated into *first-order logic* (henceforth, *FOL*) with a signature of unary predicates and the temporal order-relations. And we know from the works of (Kamp, 1968; Gabbay et al., 1980, 1994) that all *FOL* formulas over such a signature can be expressed in *LTL*. The question arises however as to whether simply composing these translations gives us the most efficient mapping from TPL^+ to *LTL* (equivalently *SVA*). In fact, we provide two different approaches for such a translation exploiting the fact that TPL^+ translates to a proper subset of *FOL*. In the first approach, we translate first TPL^+ to *LTL*. Then, we translate *LTL* to *SVA*. On the other hand, the second approach translates directly TPL^+ to *SVA*. Our translations run in linear time and avoid the non-elementary blow-up potentially entailed by the composed translations. Before giving these translations, we review some background work on the expressiveness of *FOL* and *LTL* in Section 5.1. Then, Sections 5.2 and 5.3 present the first approach which maps TPL^+ into *LTL* and then into *SVA*. In Section 5.4, we present the second approach which can map TPL^+ into *SVA* directly.

5.1 Expressiveness of FOL and LTL

This section presents earlier investigations into the translation from *FOL* to *LTL* and vice versa (see Kamp, 1968; Gabbay et al., 1980; McNaughton and Papert, 1971). We show these studies to give a better understanding of the difficulties when we translate TPL^+ to *LTL* since TPL^+ is a first-order language having variables which range over time-intervals. By contrast, the investigations mentioned above focus only on *FOL* that has variables which range over N = {0, 1, 2, . . .}, unary predicates, a binary relation symbol $<$, and one free variable. This fragment of *FOL* is known as a *monadic first-order logic*. The *monadic* term means that all relation symbols other than $<$ and $=$ contain only one variable in each formula. Note that syntax and semantics of *LTL* are introduced in Section 2.3.

Let us show how we express *LTL* formulas into *FOL*. Given *LTL* formulas "Fp" and "$U(p, q)$", we can clearly express them into *FOL* as follows, respectively:

(195) $\exists t' \; [t' \geqslant t \wedge p(t')]$

(196) $\exists t' \; [t' \geqslant t \wedge p(t') \wedge \forall t''[t \leqslant t'' < t' \rightarrow q(t'')]]$.

Formula (195) asserts that, there exists $t' \geqslant t$ such that p occurs at t', where t denotes the present time. Formula (196) asserts that, with respect to the free variable t, there exists $t' \geqslant t$ such that p occurs at t'. Moreover, q occurs for all t'' such that $t \leqslant t'' < t'$. From the above examples, It is obvious that any *LTL* formula can be translated into a *FOL* formula. This translation is linear in the size of the input.

The question then arises whether the translations can be carried out in the other direction. In fact this is the case. In Kamp (1968), Kamp proved that, over discrete linear orders, *LTL* including "strict until" and "strict since" operators is expressively complete for *FOL* with monadic predicates.

Theorem 5.1.1. *(Kamp, 1968)*
LTL with the binary temporal connectives (strict until, strict since) has the expressive power of FOL over ordered natural numbers.

Kamp's theorem does not include LTL^+ which is LTL without past operators as defined in Section 2.3. However, Gabbay in (Gabbay et al., 1980, 1994) proved that the above theorem can include LTL^+ using the separation property. This property states that, every *LTL* formula is equivalent to a Boolean combination of "pure past time", "pure present time", or "pure future time" formulas. For example, $F(q \wedge Hr)$ is not a pure formula. However, it is equivalent over linear time to the separated formula "$Hr \wedge r \wedge U(q, r)$" which is a conjunction of a pure past, a pure present and a pure future formula.

We mention, in passing, that the equivalence of *FOL* and *LTL* can be alternatively established using star-free languages (McNaughton and Papert, 1971). A star-free regular language is a subset of regular expressions which is constructed from finite languages Σ by applications of the Boolean operations + (union), $\backsim$ (complement) and $\cdot$ (concatenation dot). The syntax of star-free expressions over Σ is given by:

$$\varphi ::= \emptyset \mid \epsilon \mid a \mid \varphi \ + \ \varphi' \mid \varphi \ . \ \varphi' \mid \ \backsim \varphi$$

where $\emptyset$, ϵ, and a denote the empty set, the empty string, and constants, respectively. Notice that a ranges over all symbols in Σ. By a natural correspondence between the operations (+, $\backsim$, and $\cdot$) and the logical connectives ($\neg$, $\vee$, and $\exists$), it is easy to transform star-free expressions into *FOL* formulas. For example, over $A = \{a, b, c\}$ the expression $\backsim \emptyset \ . \ (a + c) \ . \ \backsim (\backsim \emptyset \ . \ b \ . \ \backsim \emptyset)$ defines by a *FOL* formula as follows:

(197) $\exists x((a(x) \vee c(x)) \wedge \neg\exists y(x < y \wedge b(y)))$,

This example shows that it is possible to describe star-free languages using *FOL*. In (McNaughton and Papert, 1971), McNaughton proved that any *FOL* formula can be translated into a free-star regular expression and vice versa based on the following theorem:

Theorem 5.1.2. *(McNaughton and Papert, 1971) A language is star-free if and only if it is FOL expressible (in the signature with $<$).*

Furthermore, translating star-free expressions into *LTL* is possible using counter-free DFA (stands for Deterministic Finite-state Automaton) based on Theorems 5.1.3 and 5.1.4. A counter-free DFA is defined as follows:

Definition 5.1.1. A DFA is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states; Σ is an alphabet of input symbols; $\delta : Q \times \Sigma \to Q$ is the transition function; $q_0 \in Q$ is the initial state; $F \subseteq Q$ is the set of final (or accepting) states. A DFA is said to have a counter if there exist distinct states $q_0, q_1, ..., q_n$ in $\mathcal{A}$, with $n \geq 1$, and a string $w \in \Sigma^*$, such that $\delta(q_i, w) = q_{i+1}$ for each $i \in \{0, ..., n-1\}$ and $\delta(q_n, w) = q_0$. A DFA is said to counter-free if it has no such counter, and a regular language is said to be counter-free if the minimal automaton accepting it is counter-free.

Theorem 5.1.3. *(Schützenberger, 1965) A language is counter-free DFA if and only if it is star-free expressible.*

Theorem 5.1.4. *(Wilke, 1999) A language L is expressible in LTL if and only if it is recognised by a counter-free DFA.*

Therefore, from theorems 5.1.2, 5.1.3 and 5.1.4, the equivalence of *FOL* and *LTL* also follows.

We mentioned earlier that the translation from *LTL* to *FOL* is easy and runs in linear time. However, translating *FOL* into *LTL* has some effects on the satisfiability problem in *FOL*. Since the satisfiability problem in *LTL* is PSPACE-complete, there is certainly an exponential blow-up. In fact, there is a nonelementary explosion (see Meyer, 1975).

The translation *FOL* into *LTL* is the most interesting for us because we start from TPL^+ (which can be expressed in *FOL*) and translate it to *LTL* (equivalently *SVA*). Does this mean that we have a problem? No, because we have only a subset of *FOL*. It will turn out in the next section that for our subset of *FOL* we do not have an exponential blow-up.

5.2 From TPL^+ to *LTL*

This section presents generic transformation rules generating *LTL* from TPL^+. The transformation rules are generic in the sense that the transformations can be done recursively. In providing these transformation rules, we must of course bridge the gulf between interval-based and point-based semantics, and the first question that arises is how best to do this.

The applications we have in mind—verification of protocols for clocked systems—employ models which are naturally point-based, and interpreted over the temporal frame $(\mathbb{N}, <)$. That is, time points succeed one another in a regular series of ticks, and proposition-letters are simply true or false at those time points. Thus, all evaluation will be over point-based structures of the form $\mathfrak{A} = (\mathbb{N}, <, p^{\mathfrak{A}}_{p\in P})$, and, in particular, we lose any meaningful distinction between (atomic) events and (atomic) states. How can we apply the semantics of TPL^+ (pp. 91 ff.) to such structures?

The most sensible approach is as follows. We continue to evaluate TPL^+ formulas at intervals—in this case, the domain intervals over $\mathbb{N}$. All the recursive cases in the semantics for TPL^+ are unaffected; however, the base cases must be modified, to take account of the fact that atomic events and states simply arise from the truth or falsity of proposition letters at times. More specifically, we define a truth-relation $\approx\!\!\!\models$ for TPL^+ in which the recursive clauses are identical to those for $\models$, but where the base cases are as follows.

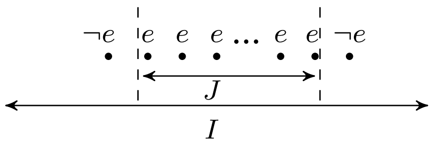
1. $\mathfrak{A} \models\!\!\!\approx_I \langle e \rangle \top$ if there is a maximal subinterval $J \subseteq I$ such that $t \in \mathfrak{A}(e)$ for all $t \in J$, and $J \subseteq I$.

2. $\mathfrak{A} \approx_I |e\rangle\top$ if $start(I) \in \mathfrak{A}(e)$.

3. $\mathfrak{A} \approx_I \langle e^{\leftleftarrows}\rangle\top$ if $start(I) \in \mathfrak{A}(e) \leftrightarrow start(I) - 1 \in \mathfrak{A}(e)$.

4. $\mathfrak{A} \approx_I [\![e]\!]$ if $t \in \mathfrak{A}(e)$ for all $t \in I$.

Note that Case 1 handles classical 'events' which take place over an extended time period. Here, we think of an a point-based structure $\mathfrak{A}$ as saying that an event occurs within some temporal context I just in case, within I, there is a continguous interval at all points of which $\mathfrak{A}$ takes the corresponding proposition letter to by true, and that this interval is maximal (is not properly contained in any other such interval). We can represent the event e occurring in Case 1 as follows:



Case 4, by contrast, handles classical 'states', which hold at points. Here, we think of an a point-based structure $\mathfrak{A}$ as saying that a state occurs throughout some temporal context I just in case, at all points of I, $\mathfrak{A}$ makes the appropriate proposition letter true. Cases 2 and 3 are similar in character to Case 4.

Now that we have discussed how to interpret TPL^+ over point-based models, the question still arises as to how to translate its formulas into LTL. After all, TPL^+-formulas are still evaluated at intervals, while LTL-formulas are evaluated at points. How can we compare the two?

Before we give an idea of how we accomplish the comparison task, let us present another difference between interval-based and point-based semantics. Consider the following sentence:

(198) Awid remains asserted until Awready goes high.

We write the corresponding TPL^+ formula using the simplification suggested in Table

4.7.2 in Section 4.7. Thus, the meaning of sentence (198) in TPL^+ is written as follows:

(199) $\{AWREADY^f\}_<[\![AWID]\!]$.

TPL^+ is evaluated over intervals which are characterized by two numerical parameters, start and end points. In contrast, *LTL* is evaluated over time points which are characterized by a single numerical parameter. To translate TPL^+ into *LTL*, we must have another point in *LTL* that represents the end point of TPL^+. Thus, we use the parameter END to represent the end point in TPL^+. This modification allows us to translate any property in TPL^+ into *LTL* using constructive rules as we shall see later. The corresponding meaning of the above TPL^+ formula in LTL is:

(200) $U(\neg END \wedge AWREADY \wedge F(END), \neg END) \wedge U(AWREADY, AWID)$.

Let us illustrate by giving the interpretations of (199) and (200) graphically and analytically. We first show graphically in Figure 5.2.1 that the interpretations of (199) and (200) can be represented in the same way.

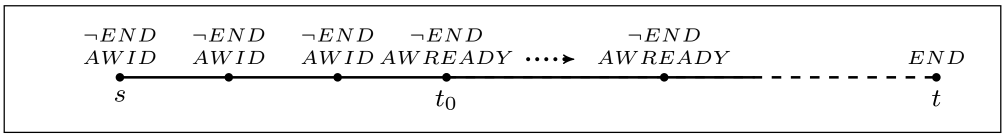


Figure 5.2.1: The interpretation of (199) and (200) graphically.

Let us analyse the above figure based on the truth condition of TPL^+ and *LTL*. Example (199) asserts that, there is a unique time point t_0 over which $AWREADY$ starts and the interval between the start point s and the unique point t_0 includes the occurrence of *AWID* at every interval. By contrast, example (200) asserts that, from the start point s of the evaluation, *AWID* must repeatedly hold until the time point before the first occurrence of *AWREADY*. Moreover, the property $\neg END$ must start holding from the beginning of the evaluation until after $AWREADY$ holds and after that the property END will eventually hold. Thus, both interpretations are clearly describing the same the truth condition, where END is taken to devote the end of the interval of TPL^+ evaluation.

Let us take another example to further observe the differences between TPL^+ and *LTL*. Consider the following sentence:

(201) Awid must be asserted within MaxWaits cycles,

where MaxWaits is a constant and its value is 16 (as discussed in Section 2.4). Let us suppose that MaxWaits = 4 instead of 16 for simplicity's sake in describing the corresponding *LTL* formula of sentence (201). The meaning of sentence (201) in TPL^+ is as follows:

(202) $|4\rangle_* \langle AWID \rangle \top$.

Again, we write the above TPL^+ formula following the simplification suggested in Table 4.7.2 in Section 4.7. We also show how we extract the meaning of "MaxWaits cycles" in TPL^+ (see example (159) on page 109). The equivalent interpretation of the above TPL^+ formula in LTL is as follows:

(203) $(AWID \vee X(AWID \vee X(AWID \vee X(AWID)))) \wedge F(END)$.

Let us now describe analytically the interpretations of (202) and (203). Example (202) asserts that, within the temporal context $[s, t]$, there is an interval J that includes the occurrence of *AWID* being true, and J is contained in $[s, s + 4]$. By contrast, example (203) asserts that, from the start point s of the evaluation, the signal *AWID* is true, and may remain true for three more clock cycles. Then, the property END will eventually hold. Again, the property END is taken to represent the end of the interval of TPL^+ evaluation.

From the above examples, TPL^+ can be expressed in *LTL* if we adopt the following approach. We suppose that we have a special atomic proposition, *END*, which signals the end of the time-period we are interested in. We define a function # such that, for any TPL^+-formula ψ any point-based structure $\mathfrak{A} = (\mathbb{N}, <, p^{\mathfrak{A}}_{p \in P})$, and any time-point $s \in \mathbb{N}, \mathfrak{A} \models_s \#(\psi, END)$ if and only if there exists some $t > s$ such that $\mathfrak{A} \models_t END$ and, taking t' to be the smallest such t for which this is the case, $\mathfrak{A} \models\!\!\approx_{[s,t']} \psi$.

For technical reasons, we take # to apply to two arguments: a TPL^+-formula (the formula that we want to translate) and an *LTL* formula picking out the end-point over evaluation. This (somewhat unusual) approach allows the transformations to be carried out recursively. Formally, the function # is given by Definition 5.2.1. The definition is devised to secure the following theorem:

Theorem 5.2.1. *For all TPL^+-sentences ψ, all LTL sentences, π all point-based structures $\mathfrak{A} = (\mathbb{N}, <, p^{\mathfrak{A}}_{p\in P})$, and all $s \in \mathbb{N}$, $\mathfrak{A} \models_s \#(\psi, \pi)$ if and only if there exists some $t > s$ such that $\mathfrak{A} \models_t \pi$ and, taking t' to be the smallest such t for which this is the case, $\mathfrak{A} \models\!\!\approx_{[s,t']} \psi$.*

Definition 5.2.1. We define a function

$\#\colon TPL^+ \times LTL \to LTL$

by recursion on the structure of the first argument as follows.

$$\left.\begin{array}{ll}
\text{(T1)} & \#(\langle e\rangle\top, \pi) = U(X^{\text{-}1}\neg e \wedge e \wedge U(\neg e, \neg\pi), \neg\pi) \wedge F(\pi). \\[1.5em]
\text{(T2)} & \#(|e\rangle\top, \pi) = U(\neg e \wedge \neg\pi, e \wedge \neg\pi) \wedge F(\pi). \\[1.5em]
\text{(T3)} & \#([\![e]\!], \pi) = U(\pi, e). \\[1.5em]
\text{(T4)} & \#(\langle e^{\underline{\leftharpoonup}}\rangle\top, \pi) = (e \leftrightarrow X^{\text{-}1}e) \wedge F(\pi). \\[1.5em]
\text{(T5)} & \#(|n\rangle_*[\![e]\!], \pi) = (e \wedge (Xe \wedge (XXe\wedge\ {}^{n\ \underset{\cdots\cdot}{times}}\ ))) \wedge F(\pi). \\[1.5em]
\text{(T6)} & \#(|n\rangle_*\langle e\rangle\top, \pi) = (e \vee X(e \vee X(e\vee\ {}^{n\ \underset{\cdots\cdot}{times}}\ ))) \wedge F(\pi).
\end{array}\right\}\ \text{Base Cases}$$

(T7) $\#(\{e^f\}\psi, \pi) = U(e \wedge\ \#(\psi, \neg e) \wedge U(\neg e, \neg\pi), \neg\pi) \wedge U(\pi, \neg e)$.

(T8) $\#([e]\psi, \pi) = U(\pi, e \wedge \#(\psi, \neg e))$.

(T9) $\#(\{e^f\}_<\psi, \pi) = U(\neg\pi \wedge e \wedge F(\pi), \neg\pi) \wedge \#(\psi, e)$.

(T10) $\#(\{e^l\}_{>}\psi, \pi) = U(e \wedge U(\neg e \wedge \#(\psi, \pi) \wedge U(\pi, \neg e), \neg\pi \wedge e), \neg\pi)$.

(T11) $\#([e]_{>}\psi, \pi) = U(\pi, e \rightarrow X(\#(\psi, \top) \vee \pi))$.

(T12) $\#([e]_{<}\psi, \pi) = U(\pi, e \rightarrow X^{-1}(\neg e \wedge \#(\psi, e)))$.

(T13) $\#(\{e^f\}_{<+}\psi, \pi) = U(\neg\pi \wedge e \wedge F(\pi), \neg\pi) \wedge \#(\psi, e \wedge X\neg e \wedge \neg\pi)$.

We now turn to the proof of Theorem 5.2.1 which states that any TPL^+ formula can be mapped into an LTL formula using transformation rules that appear in Definition 5.2.1.

Proof of Theorem 5.2.1. We proceed by induction on the structure of ϕ.

Base case (1): $\phi = \langle e \rangle \top$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#(\langle e \rangle \top, \pi)$ i.e. $\mathfrak{A} \models_s U(X^{-1}\neg e \wedge e \wedge U(\neg e, \neg\pi), \neg\pi) \wedge F(\pi)$. From the conjunct $F(\pi)$, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From the first conjunct, there exists a $t_0 \geq s$ such that $\mathfrak{A} \models_{t_0} X^{-1}\neg e \wedge e \wedge U(\neg e, \neg\pi)$, and moreover, $\neg\pi$ is true at every time point from s to t_0. Therefore $t_0 \leq t'$. Let take t_0 to be the first such time. Now $\mathfrak{A} \models_{t_0} X^{-1}\neg e \wedge e \wedge U(\neg e, \neg\pi)$ so there exists a time point $t_1 \geq t_0$ such that $\mathfrak{A} \models_{t_1} \neg e$, and moreover $\neg\pi$ holds at all points from t_0 to t_1-1 inclusive, whence $t_1 \leq t'$. We may as well take t_1 to be the smallest such point. So $[t_0, t_1]$ is a maximal interval J such that $e^{\mathfrak{A}}$ for every $t \in J$. Therefore, $\mathfrak{A} \approx\!\!\!\!\models_{[s,t']} \langle e \rangle \top$ for any I such that $J \subseteq I$ and $I = [s, t']$, i.e. $\mathfrak{A} \approx\!\!\!\!\models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Certainly, then we have $\mathfrak{A} \models_s F(\pi)$. Moreover, suppose $\mathfrak{A} \approx\!\!\!\!\models_{[s,t']} \langle e \rangle \top$. From the semantics of $\langle e \rangle \top$, for some $J \subseteq [s, t'], J \in \mathfrak{A}(e)$. Furthermore, we take J to be a maximal subinterval of $e^{\mathfrak{A}}$. Hence, $\mathfrak{A} \models_s U(X^{-1}\neg e \wedge e \wedge U(\neg e, \neg\pi), \neg\pi)$. So we have $\mathfrak{A} \models_s \#(\langle e \rangle \top, \pi)$ (see Figure 5.2.2).

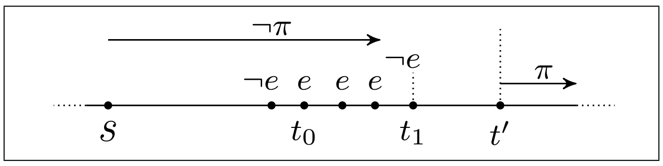


Figure 5.2.2: Illustrating the proof of rule (T1).

Base case (2): $\phi = |e\rangle\top$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#(|e\rangle\top, \pi)$ i.e. $\mathfrak{A} \models_s U(\neg e \wedge \neg\pi, e \wedge \neg\pi) \wedge F(\pi)$. From the conjunct $F(\pi)$, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. From the first conjunct, $\mathfrak{A} \models_s U(\neg e \wedge \neg\pi, e \wedge \neg\pi)$. Moreover, there exists a time point $t_0 \geq s$ such that $\mathfrak{A} \models_{t_0} \neg e \wedge \neg\pi$. So by semantics of TPL^+, $\mathfrak{A} \approx_{[s,t']} |e\rangle\top$, i.e. $\mathfrak{A} \approx_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Certainly, then we have $\mathfrak{A} \models_s F(\pi)$. Moreover, suppose $\mathfrak{A} \approx_{[s,t']} |e\rangle\top$. From the semantics of $|e\rangle\top$, for some $J, J \in \mathfrak{A}(e)$ and J is the initial interval of $[s, t']$. Hence, $\mathfrak{A} \models_s U(\neg e \wedge \neg\pi, e \wedge \neg\pi)$. So we have $\mathfrak{A} \models_s \#(|e\rangle\top, \pi)$ (see Figure 5.2.3).

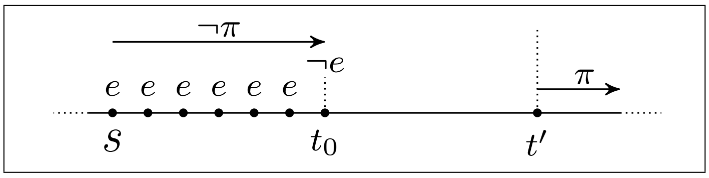


Figure 5.2.3: Illustrating the proof of rule (T2).

Base case (3): $\phi = [\![e]\!]$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#([\![e]\!], \pi)$ i.e. $\mathfrak{A} \models_s U(\pi, e)$. We see there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. Moreover, for all $t_0 \geq s$ such that $\mathfrak{A} \models_{t_0} e$. So by semantics of TPL^+, $\mathfrak{A} \approx_{[s,t']} [\![e]\!]$, i.e. $\mathfrak{A} \approx_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$, and suppose $\mathfrak{A} \approx_{[s,t']} [\![e]\!]$. From the semantics of $[\![e]\!]$, for all $J \subseteq [s, t']$ such that $J \in \mathfrak{A}(e)$. Hence, $\mathfrak{A} \models_s U(\pi, e)$. So we have $\mathfrak{A} \models_s \#([\![e]\!], \pi)$ (see Figure 5.2.4).

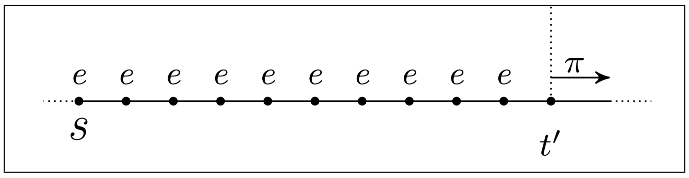


Figure 5.2.4: Illustrating the proof of rule (T3).

Base case (4): $\phi = \langle e^{\Leftarrow}\rangle\top$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#(\langle e^{\Leftarrow}\rangle\top, \pi)$ i.e. $\mathfrak{A} \models_s (e \leftrightarrow X^{-1}e) \wedge F(\pi)$. From the second conjunct, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From the first conjunct, $\mathfrak{A} \models_s e \Leftrightarrow \mathfrak{A} \models_{s-1} e$. So by semantics of TPL^+, $\mathfrak{A} \approx_{[s,t']} \langle e^{\Leftarrow}\rangle\top$, i.e. $\mathfrak{A} \approx_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Certainly, then we have $\mathfrak{A} \models_s F(\pi)$. Moreover, suppose $\mathfrak{A} \approx_{[s,t']} \langle e^{\leftleftarrows}\rangle\top$. From the semantics of $\langle e^{\leftleftarrows}\rangle\top$, since $start([s,t']) = s$, $\mathfrak{A} \models_{[s,s]} e \Leftrightarrow \mathfrak{A} \models_{[s\text{-}1,s\text{-}1]} e$. Hence, $\mathfrak{A} \models_s e \Leftrightarrow \mathfrak{A} \models_{s\text{-}1} e$ which is equivalent to $\mathfrak{A} \models_s (e \leftrightarrow X^{-1}e)$. So we have $\mathfrak{A} \models_s \#(\langle e^{\leftleftarrows}\rangle\top, \pi)$ (see Figure 5.2.5).

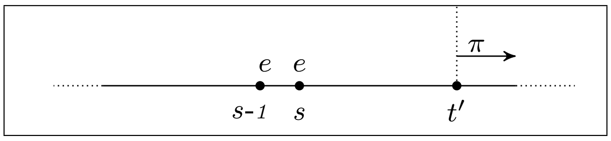


Figure 5.2.5: Illustrating the proof of rule (T4).

Base case (5): $\phi = |n\rangle_*[\![e]\!]$. Suppose $\mathfrak{A} \models_s \#(\phi,\pi)$ and n=4. i.e. $\mathfrak{A} \models_s \#(|4\rangle_*[\![e]\!],\pi)$ i.e. $\mathfrak{A} \models_s (e \wedge (Xe \wedge (XXe \wedge (XXXe)))) \wedge F(\pi)$. From the second conjunct, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From the first conjunct, $\mathfrak{A} \models_s (e \wedge (Xe \wedge (XXe \wedge (XXXe))))$ where e must hold for four consecutive points starting from s. So by semantics of TPL^{+} in Section 4.4, $\mathfrak{A} \models_{[s,t']} |4\rangle_*[\![e]\!]$ which derived from $\mathfrak{A} \models_{[s,t']} |4\rangle_*\psi$ and $\mathfrak{A} \approx_{[s,t']} [\![e]\!]$, i.e. $\mathfrak{A} \models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Certainly, then we have $\mathfrak{A} \models_s F(\pi)$. Moreover, suppose $\mathfrak{A} \models_{[s,t']} |4\rangle_*[\![e]\!]$. From the semantics of $|4\rangle_*[\![e]\!]$, $\mathfrak{A} \approx_{[s,s+4]} [\![e]\!]$, and from the semantics of $[\![e]\!]$, for all $J \subseteq [s, s{+}4]$ such that $J \in \mathfrak{A}(e)$. Hence, $\mathfrak{A} \models_s (e \wedge (Xe \wedge (XXe \wedge (XXXe))))$. So we have $\mathfrak{A} \models_s \#(|4\rangle_*[\![e]\!],\pi)$ (see Figure 5.2.6).

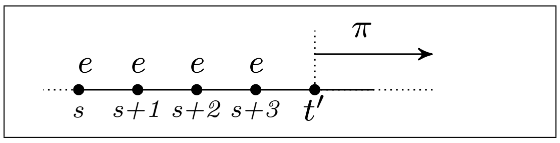


Figure 5.2.6: Illustrating the proof of rule (T5).

Base case (6): $\phi = |n\rangle_*\langle e\rangle\top$. Suppose $\mathfrak{A} \models_s \#(\phi,\pi)$ and n=4. i.e. $\mathfrak{A} \models_s \#(|4\rangle_*\langle e\rangle\top,\pi)$ i.e. $\mathfrak{A} \models_s (e \vee X(e \vee X(e \vee X(e)))) \wedge F(\pi)$. From the second conjunct, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From the first conjunct, $\mathfrak{A} \models_s (e \vee X(e \vee X(e \vee X(e))))$, where e may hold at any point from s to $s{+}4$. So by semantics of TPL^{+} in Section 4.4, $\mathfrak{A} \models_{[s,t']} |4\rangle_*\langle e\rangle\top$ which derived from $\mathfrak{A} \models_{[s,t']} |4\rangle_*\psi$ and $\mathfrak{A} \approx_{[s,t']} \langle e\rangle\top$, i.e. $\mathfrak{A} \models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Certainly, then we have $\mathfrak{A} \models_s F(\pi)$. Moreover, suppose $\mathfrak{A} \models_{[s,t']} |4\rangle_* \langle e \rangle \top$. From the semantics of $|4\rangle_* \langle e \rangle \top$, $\mathfrak{A} \approx_{[s,s+4]} \langle e \rangle \top$, and from the semantics of $\langle e \rangle \top$, for some $J \subseteq [s, s+4]$, $J \in \mathfrak{B}(e)$. Furthermore, we take J to be a maximal subinterval of $e^{\mathfrak{B}}$. Hence, $\mathfrak{A} \models_s (e \vee X(e \vee X(e \vee X(e))))$. So we have $\mathfrak{A} \models_s \#(|4\rangle_* \langle e \rangle \top, \pi)$.

Recursive case (1): $\phi = \{e^f\}\psi$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#(\{e^f\}\psi, \pi)$ i.e. $\mathfrak{A} \models_s U(e \wedge \#(\psi, \neg e) \wedge U(\neg e, \neg\pi), \neg\pi) \wedge U(\pi, \neg e)$. We see first there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From $e \wedge \#(\psi, \neg e) \wedge U(\neg e, \neg\pi)$, there exists a t_0 such that $\mathfrak{A} \models_{t_0} e \wedge \#(\psi, \neg e) \wedge U(\neg e, \neg\pi)$, and moreover, $\neg\pi$ is true at every time point from s to t_0. Therefore $t_0 \leq t'$. By the induction hypothesis, there exists a time point $t_1 \geq t_0$ such that $\mathfrak{A} \models_{t_1} \neg e$, and $\mathfrak{A} \models_{[t_0,t_1]} \psi$. From $U(\pi, \neg e)$, we see that $\neg e$ must remain true until t'-1. So by semantics of TPL^+ in Section 4.4, $\mathfrak{A} \models_{[s,t']} \{e^f\}\psi$ which derived from $\mathfrak{A} \models_{[s,t']} \{\alpha\}\psi$ and $\mathfrak{A} \models_{[s,t']} e^f$, i.e. $\mathfrak{A} \models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$, and suppose $\mathfrak{A} \models_{[s,t']} \{e^f\}\psi$. From the semantics of $\mathfrak{A} \models_{[s,t']} \{e^f\}\psi$, there is a unique $J \subseteq [s, t']$ such that $J \in \mathfrak{A}(e)$, J is the first such interval, and for that $J, \mathfrak{A} \models_J \psi$. Hence, $\mathfrak{A} \models_s U(e \wedge \#(\psi, \neg e) \wedge U(\neg e, \neg\pi), \neg\pi) \wedge U(\pi, \neg e)$. So we have $\mathfrak{A} \models_s \#(\{e^f\}\psi, \pi)$ (see Figure 5.2.7).

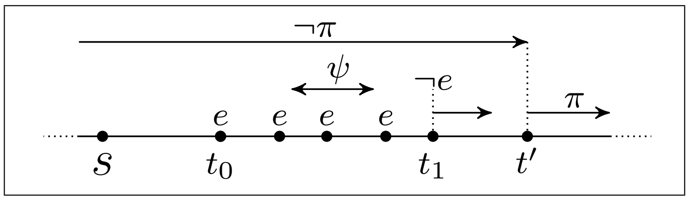


Figure 5.2.7: Illustrating the proof of rule (T7).

Recursive case (2): $\phi = [e]\psi$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#([e]\psi, \pi)$ i.e. $\mathfrak{A} \models_s U(\pi, e \wedge \#(\psi, \neg e))$. We see first there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From $e \wedge \#(\psi, \neg e)$, for all $t_0 \geq s$ such that $\mathfrak{A} \models_{t_0} e \wedge \#(\psi, \neg e)$. By the induction hypothesis, for all $t_1 \geq t_0$ such that $\mathfrak{A} \models_{t_1} \neg e$ and $\mathfrak{A} \models_{[t_0,t_1]} \psi$. So by semantics of TPL^+ in Section 4.4, $\mathfrak{A} \models_{[s,t']} [e]\psi$, i.e. $\mathfrak{A} \models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$, and suppose $\mathfrak{A} \models_{[s,t']} [e]\psi$. From the semantics of $[e]\psi$ in Section 4.4, for all $J \subseteq [s, t'\text{-}1], J \in \mathfrak{A}(e)$ implies $\mathfrak{A} \models_J \psi$. Hence, $\mathfrak{A} \models_s U(\pi, e \wedge \#(\psi, \neg e))$. So we have $\mathfrak{A} \models_s \#([e]\psi, \pi)$ (see Figure 5.2.8).

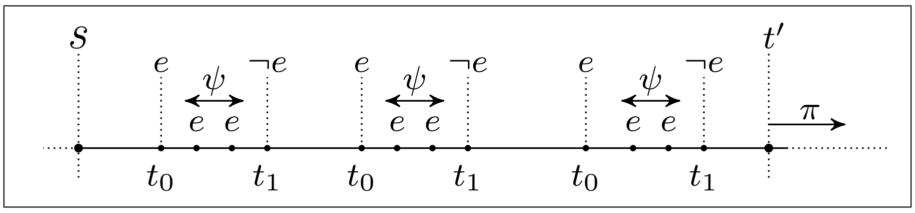


Figure 5.2.8: Illustrating the proof of rule (T8).

Recursive case (3): $\phi = \{e^f\}_{<}\psi$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#(\{e^f\}_{<}\psi, \pi)$ i.e. $\mathfrak{A} \models_s U(\neg\pi \wedge e \wedge F(\pi), \neg\pi) \wedge \#(\psi, e)$. From the conjunct $F(\pi)$, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From the first conjunct, there exists a $t_0 \geq s$ such that $\mathfrak{A} \models_{t_0} \neg\pi \wedge e \wedge F(\pi)$, and moreover $\neg\pi$ is true at every time point from s to t_0. By the induction hypothesis on the second conjunct, we have $\mathfrak{A} \models_{[s,t_0\text{-}1]} \psi$. So by semantics of TPL^+ in Section 4.4, $\mathfrak{A} \models_{[s,t']} \{e^f\}_{<}\psi$ which derived from $\mathfrak{A} \models_{[s,t']} \{\alpha\}_{<}\psi$ and $\mathfrak{A} \models_{[s,t']} e^f$, i.e. $\mathfrak{A} \models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Certainly, then we have $\mathfrak{A} \models_s F(\pi)$. Moreover, suppose $\mathfrak{A} \models_{[s,t']} \{e^f\}_{<}\psi$. From the semantics of $\{e^f\}_{<}\psi$, there is a unique $J \subseteq [s, t']$ such that $J \in \mathfrak{A}(e)$, J is the first such interval, and for that J, $\mathfrak{A} \models_{init(J,[s,t'])} \psi$. Hence, $\mathfrak{A} \models_s U(\neg\pi \wedge e \wedge F(\pi), \neg\pi) \wedge \#(\psi, e)$. So we have $\mathfrak{A} \models_s \#(\{e^f\}_{<}\psi, \pi)$ (see Figure 5.2.9).

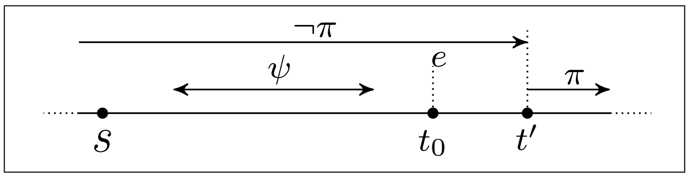


Figure 5.2.9: Illustrating the proof of rule (T9).

Recursive case (4): $\phi = \{e^l\}_{>}\psi$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#(\{e^l\}_{>}\psi, \pi)$ i.e. $\mathfrak{A} \models_s U(e \wedge U(\neg e \wedge \#(\psi, \pi) \wedge U(\pi, \neg e), \neg\pi \wedge e), \neg\pi)$. From π, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From $e \wedge U(\neg e \wedge \#(\psi, \pi) \wedge U(\pi, \neg e), \neg\pi \wedge e)$, there exists a $t_0 \geq s$ such that $\mathfrak{A} \models_{t_0} e \wedge U(\neg e \wedge \#(\psi, \pi) \wedge U(\pi, \neg e), \neg\pi \wedge e)$, and moreover $\neg\pi$ is true at every time point from s to t_0. By the induction hypothesis, there exists $t_1 \geq t_0$ such that $\mathfrak{A} \models_{t_1} \neg e \wedge \#(\psi, \pi)$. From $U(\pi, \neg e)$, we see that $\neg e$ must remain true until t'-1. So by semantics of TPL^+ in Section 4.4, $\mathfrak{A} \models_{[s,t']} \{e^l\}_{>}\psi$ which derived from $\mathfrak{A} \models_{[s,t']} \{\alpha\}_{>}\psi$ and $\mathfrak{A} \models_{[s,t']} e^l$, i.e. $\mathfrak{A} \models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Moreover, suppose $\mathfrak{A} \models_{[s,t']} \{e^l\}_{>}\psi$. From the semantics of $\{e^l\}_{>}\psi$, there is a unique $J \subseteq [s, t']$ such that $J \in \mathfrak{A}(e)$, J is the last such interval, and for that J, $\mathfrak{A} \models_{fin(J,[s,t'])} \psi$. Hence, $\mathfrak{A} \models_s U(e \wedge U(\neg e \wedge \#(\psi, \pi) \wedge U(\pi, \neg e), \neg\pi \wedge e), \neg\pi)$. So we have $\mathfrak{A} \models_s \#(\{e^l\}_{>}\psi, \pi)$ (see Figure 5.2.10).

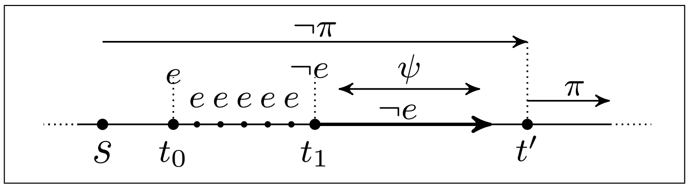


Figure 5.2.10: Illustrating the proof of rule (T10).

Recursive case (5): $\phi = [e]_{>}\psi$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#([e]_{>}\psi, \pi)$ i.e. $\mathfrak{A} \models_s U(\pi, e \to X(\#(\psi, \top) \vee \pi))$. From π, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From $e \to X(\#(\psi, \top) \vee \pi)$, for all $t_0 \geq s$ such that $\mathfrak{A} \models_{t_0} e \to X(\#(\psi, \top) \vee \pi)$. By the induction hypothesis, for all t_0+1 such that either $\mathfrak{A} \models_{t_0+1} \#(\psi, \top)$ or $\mathfrak{A} \models_{t_0+1} \pi$. So by the semantics of TPL^+ in Section 4.4, $\mathfrak{A} \models_{[s,t']} [e]_{>}\psi$, i.e. $\mathfrak{A} \models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Moreover, suppose $\mathfrak{A} \approx_{[s,t']} [e]_{>}\psi$. From the semantics of $[e]_{>}\psi$, for all $J \subseteq [s, t']$ such that $J \in \mathfrak{A}(e)$ implies $\mathfrak{A} \models_{[end(J)+1,end(J)+1]} \psi$. Hence, $\mathfrak{A} \models_s U(\pi, e \to X(\#(\psi, \top) \vee \pi))$. So we have $\mathfrak{A} \models_s \#([e]_{>}\psi, \pi)$ (see Figure 5.2.11).

Recursive case (6): $\phi = [e]_{<}\psi$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#([e]_{<}\psi, \pi)$ i.e. $\mathfrak{A} \models_s U(\pi, e \to X^{-1}(\neg e \wedge \#(\psi, e)))$. From π, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From $e \to X^{-1}(\neg e \wedge \#(\psi, e))$, for all $t_0 \geq s$ such

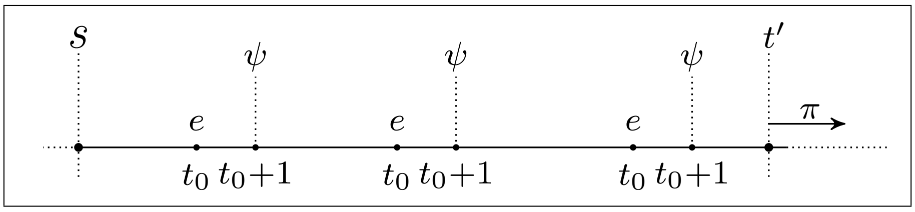


Figure 5.2.11: Illustrating the proof of rule (T11).

that $\mathfrak{A} \models_{t_0} e \to X^{\text{-}1}(\neg e \wedge \#(\psi, \pi))$. By the induction hypothesis, for all t_0-1 such that $\mathfrak{A} \models_{t_0\text{-}1} \neg e \wedge \#(\psi, \pi)$. So by semantics of TPL^+ in Section 4.4, $\mathfrak{A} \models_{[s,t']} [e]_< \psi$, i.e. $\mathfrak{A} \models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Moreover, suppose $\mathfrak{A} \models_{[s,t']} [e]_< \psi$. From the semantics of $[e]_< \psi$, for all $J \subseteq [s, t']$ such that $J \in \mathfrak{A}(e)$ implies $\mathfrak{A} \models_{[start(J)\text{-}1, start(J)\text{-}1]} \psi$. Hence, $\mathfrak{A} \models_s U(\pi, e \to X^{\text{-}1}(\neg e \wedge \#(\psi, e)))$. So we have $\mathfrak{A} \models_s \#([e]_< \psi, \pi)$ (see Figure 5.2.12).

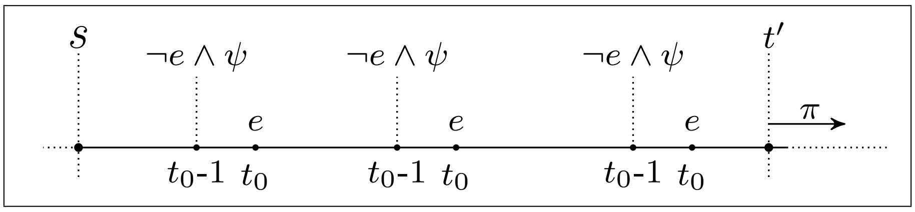


Figure 5.2.12: Illustrating the proof of rule (T12).

Recursive case (7): $\phi = \{e^f\}_{<+}\psi$. Suppose $\mathfrak{A} \models_s \#(\phi, \pi)$. i.e. $\mathfrak{A} \models_s \#(\{e^f\}_{<+}\psi, \pi)$ i.e. $\mathfrak{A} \models_s U(\neg\pi \wedge e \wedge F(\pi), \neg\pi) \wedge \#(\psi, e \wedge X\neg e \wedge \neg\pi)$. From $F(\pi)$, there exists $t \geq s$ such that $\mathfrak{A} \models_t \pi$. Take t' to be the smallest such t. From the first conjunct, there exists a $t_0 \geq s$ such that $\mathfrak{A} \models_{t_0} \neg\pi \wedge e \wedge F(\pi)$, and moreover $\neg\pi$ is true at every time point from s to t_0. By the induction hypothesis on the second conjunct, there exists a time point $t_1 \geq t_0$ such that $\mathfrak{A} \models_{[s,t_1]} \psi$. So by the semantics of TPL^+ in Section 4.4, $\mathfrak{A} \models_{[s,t']} \{e^f\}_{<+}\psi$ which derived from $\mathfrak{A} \models_{[s,t']} \{\alpha\}_{<+}\psi$ and $\mathfrak{A} \models_{[s,t']} e^f$, i.e. $\mathfrak{A} \models_{[s,t']} \phi$.

Conversely, suppose that t' is the smallest number $> s$ such that $\mathfrak{A} \models_t \pi$. Certainly, then we have $\mathfrak{A} \models_s F(\pi)$. Moreover, suppose $\mathfrak{A} \models_{[s,t']} \{e^f\}_{<+}\psi$. From the semantics of $\{e^f\}_{<+}\psi$, there is a unique $J \subseteq [s, t']$ such that $J \in \mathfrak{A}(e)$, J is the first such interval, and for that J, $\mathfrak{A} \models_{[s,end(J)+1]} \psi$, where start($I$) = s. Hence, $\mathfrak{A} \models_s U(\neg\pi \wedge e \wedge F(\pi), \neg\pi) \wedge$

$\#(\psi, e \wedge X\neg e \wedge \neg\pi)$. So we have $\mathfrak{A} \models_s \#(\{e^f\}_{<+}\psi, \pi)$ (see Figure 5.2.13).

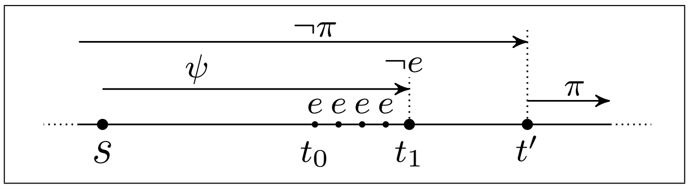


Figure 5.2.13: Illustrating the proof of rule (T13).

By Theorem 5.2.1, any TPL^+ can be transformed into a satisfactory LTL formula using the transformation rules in Definition 5.2.1. □

Now, we conclude this section with some examples illustrating the mapping procedures. Examples (5.2.2)—(5.2.6) have TempCNL sentences that are interpreted in TPL^+ formulas. We map these TPL^+ formulas into LTL using our rules in Definition 5.2.1.

Example 5.2.1. Awid must be asserted within (MaxWaits/4) cycles.

$$\#(|4\rangle_*\langle AWID\rangle\top, END) = X(AWID \vee (AWID \wedge XAWID)) \wedge F(END)\,\text{(T6)}$$

Example 5.2.2. Awid remains asserted until Awready goes high.

$$\begin{aligned}\#(\{AWREADY^f\}_<[\![AWID]\!], END) &= U(\neg END \wedge AWREADY \wedge F(END), \neg END)\wedge\\ &\quad \#([\![AWID]\!], AWREADY) && \text{(T9)}\\ &= U(\neg END \wedge AWREADY \wedge F(END), \neg END)\wedge\\ &\quad U(AWREADY, AWID) && \text{(T3)}\end{aligned}$$

Example 5.2.3. Awid must be stable once Awvalid goes high.

$$\begin{aligned}\#([AWVALID]_>\langle AWID^{\Leftarrow}\rangle\top, END) &= U(END, AWVALID \rightarrow\\ &\quad X(\#(\langle AWID^{\Leftarrow}\rangle\top, \top) \vee END)) && \text{(T11)}\\ &= U(END, AWVALID \rightarrow\\ &\quad X(((AWID \leftrightarrow X^{-1}AWID) \wedge F(\top)) \vee END)) && \text{(T4)}\end{aligned}$$

Example 5.2.4. Bvalid must remain low until after Wlast becomes high.

$$\begin{aligned}\#(\{WLAST^{f}\}_{<+}[\![\neg BVALID]\!], END) &= U(\neg END \wedge WLAST \wedge F(END), \neg END)\wedge \\ &\quad \#([\![\neg BVALID]\!], WLAST \wedge X\neg WLAST \wedge \neg END) \quad \text{(T13)} \\ &= U(\neg END \wedge WLAST \wedge F(END), \neg END)\wedge \\ &\quad U(WLAST \wedge X\neg WLAST \wedge \neg END, \neg BVALID) \quad \text{(T3)}\end{aligned}$$

Note that each of the above examples have mapped to LTL using only one transformation rule or two transformation rules. Both TPL^{+} and LTL formulas in each example show that temporal properties in TempCNL sentences are expressed in the same way in both formulas. Thus, mapping TPL^{+} into LTL using our rules are correct in terms of both formulas sharing the same truth-condition.

Let us examine TempCNL sentences which have more than two temporal prepositions, and how we map their TPL^{+} formulas into LTL using our rules.

Example 5.2.5. After Awvalid goes high, Awid must be high for three cycles.

$$\begin{aligned}\#(\{AWVALID^{l}\}_{>}|3\rangle_{*}[\![AWID]\!], END) &= U(AWVALID\wedge \\ &\quad U(\neg AWVALID \\ &\quad\quad \#(|3\rangle_{*}[\![AWID]\!], END)\wedge \\ &\quad\quad U(END, \neg AWVALID), \\ &\quad\quad\quad \neg END \wedge AWVALID), \neg END) \quad \text{(T10)} \\ &= U(AWVALID\wedge \\ &\quad U(\neg AWVALID\wedge \\ &\quad\quad (AWID \wedge (X\ AWID \wedge XX\ AWID))\wedge \\ &\quad\quad F(END) \wedge U(END, \neg AWVALID), \\ &\quad\quad\quad \neg END \wedge AWVALID), \neg END) \quad \text{(T5)}\end{aligned}$$

Example 5.2.6. After Awvalid goes high, Awid must remain high until Awready becomes high.

$$\begin{aligned}
\#(\{AWVALID^{l}\}_{>}\{AWREADY^{f}\}_{<}[\![AWID]\!], END) &= U(AWVALID\wedge \\
&\quad U(\neg AWVALID\wedge \\
&\qquad \#(\{AWREADY^{f}\}_{<}[\![AWID]\!], END)\wedge \\
&\qquad\quad U(END, \neg AWVALID), \\
&\qquad\qquad \neg END \wedge AWVALID), \neg END) \qquad \text{(T10)} \\
&= U(AWVALID\wedge \\
&\quad U(\neg AWVALID\wedge \\
&\qquad U(\neg END \wedge AWREADY \wedge F(END), \neg END)\wedge \\
&\qquad\quad \#([\![AWID]\!], AWREADY)\wedge \\
&\qquad\qquad U(END, \neg AWVALID), \\
&\qquad\qquad\quad \neg END \wedge AWVALID), \neg END) \qquad \text{(T9)} \\
&= U(AWVALID\wedge \\
&\quad U(\neg AWVALID\wedge \\
&\qquad U(\neg END \wedge AWREADY \wedge F(END), \neg END)\wedge \\
&\qquad\quad U(AWREADY, AWID)\wedge \\
&\qquad\qquad U(END, \neg AWVALID), \\
&\qquad\qquad\quad \neg END \wedge AWVALID), \neg END) \qquad \text{(T3)}
\end{aligned}$$

Each of the above examples has been mapped to LTL using three transformation rules. Both TPL^{+} and LTL formulas in each example show again that temporal properties in TempCNL sentences are expressed in the same way in both formulas. Thus, by applying our transformation rules, we can extract LTL from TPL^{+} in a concise and efficient way. It is worth pointing out that English sentences often do not include more than three temporal prepositions. Nevertheless, our transformation rules are robust at mapping TPL^{+} formulas possessing more than three depth levels into LTL. To conclude, we have shown that generating LTL from TPL^{+} is theoretically possible using our transformation rules as illustrated using the examples (5.2.2)—(5.2.6).

5.3 From *LTL* to *SVA*

In the previous section, we showed how we can map TPL^+ into *LTL* using that transformation rules listed in Definition 5.2.1, and moreover we proved theoretically the equivalent between TPL^+ and *LTL* in those rules. Now, we will show how we translate *LTL* to *SVA*. However, first we show how the translation between *LTL* and *SVA* was developed historically.

To begin with, model checking had been having an increasing industrial impact since Pnueli (1977)'s proposal. Pnueli used *LTL* to specify and verify the correctness of concurrent programs. However, *LTL* was not expressive enough at that time for this task. This problem was mentioned in Wolper (1983) who pointed that there are specific ω-regular events that cannot be expressed in *LTL*.

In 1995, Intel engineers started the development of industrial temporal logics for verifying properties of hardware designs. The first generation of Intel formal verification tools was called Prover (see Kamhi et al., 1997). Prover was later enhanced by BDD-based model checker (developed in McMillan (1993)) and a new specification language, called *FSL* which was an *LTL*-based. *FSL* was model checked with the automata-theoretic technique in (Vardi and Wolper, 1986). Comparing *FSL* with *LTL* in industrial usage, *FSL* can express more properties than *LTL*.

Later, Intel released a new generation of the specification language, called *ForSpec Temporal logic* (FTL) (see Armoni et al., 2002). *FTL* is basically a combination of *LTL*, regular expressions, and some features corresponding to clocks and resets (which comprises a core of many specification languages). *FTL* has the full expressive power of ω-regular expression.

In 2003, Accellera Formal Verification Technical Committee chose some languages (such as Sugar from IBM, ForSpec from Intel, CBV from Motorola, OpenVera from Synopsys, etc.) to build an IEEE standard for formal verification language. As result, *SVA* was released in IEEE Std 1800-2005 (2005). However, ForSpec was not embedded until the next version of *SVA* which was published at IEEE Std 1800-2009 (2009). Then, *SVA* has all *LTL* operators in which *SVA* have the same expressive power of *LTL*. The final version of *SVA* is described in IEEE Std 1800-2012 (2013).

Now, since the latest version of *SVA* includes *LTL* operators, the translation from *LTL* to *SVA* becomes a straightforward task. In Table 5.3.1, we show all standard *LTL* operators and their equivalent operators in *SVA*.

Unary operators	Binary operators	Other useful constructions
X $\equiv$ nexttime	U $\equiv$ s_until	$e \leftrightarrow X^{-1}e \equiv$ \$stable(e)
F $\equiv$ s_eventually	$\rightarrow \equiv$ implies	$X^{-1}\neg e \wedge e \equiv$ \$rise(e)
G $\equiv$ always	$\wedge \equiv$ and	$X^{-1}e \wedge \neg e \equiv$ \$fell(e)
$\neg \equiv$ not	$\vee \equiv$ or	$X^{-1}e \equiv$ \$past(e,1)

Table 5.3.1: *LTL* operators and their equivalent operators in *SVA*.

As shown in Table 5.3.1, the translations from *LTL* to *SVA* are easy tasks since any *LTL* formula can be expressed in *SVA* using the equivalent operators.

Let us now show how we can translate *LTL* formulas of sentences (204) and (205) into *SVA* with *LTL* operators. Note that sentence (198) is repeated here as (204) . Examples 5.4.1 and 5.4.2 on page 161 show the steps of translating *LTL* formulas into *SVA* with *LTL* operators using the equivalent operators in Table 5.3.1.

(204) Awid remains asserted until Awready goes high

(205) Awid must be stable once Awvalid goes high.

Example 5.4.1.

1. $f_{LTL\rightarrow SVA}(U(\neg END \wedge AWREADY \wedge F(END), \neg END) \wedge$
   $U(AWREADY, AWID))$ — $(\wedge \equiv \mathit{and})$

2. $f_{LTL\rightarrow SVA}(U(\neg END \wedge AWREADY \wedge F(END), \neg END))\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — $(U \equiv \mathit{s\_until})$

3. $\mathit{s\_until}(f_{LTL\rightarrow SVA}(\neg END \wedge AWREADY \wedge F(END)), f_{LTL\rightarrow SVA}(\neg END))\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — $(\wedge \equiv \mathit{and})$

4. $\mathit{s\_until}(f_{LTL\rightarrow SVA}(\neg END)\ \mathit{and}\ f_{LTL\rightarrow SVA}(AWREADY)\ \mathit{and}\ f_{LTL\rightarrow SVA}(F(END)), f_{LTL\rightarrow SVA}(\neg END))\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — $(\neg \equiv \mathit{not})$

5. $\mathit{s\_until}(\mathit{not}\ f_{LTL\rightarrow SVA}(END)\ \mathit{and}\ f_{LTL\rightarrow SVA}(AWREADY)\ \mathit{and}\ f_{LTL\rightarrow SVA}(F(END)), f_{LTL\rightarrow SVA}(\neg END))\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — atomic propositions similarly interpreted

5. $\mathit{s\_until}(\mathit{not}\ END\ \mathit{and}\ f_{LTL\rightarrow SVA}(AWREADY)\ \mathit{and}\ f_{LTL\rightarrow SVA}(F(END)), f_{LTL\rightarrow SVA}(\neg END))\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — atomic propositions similarly interpreted

6. $\mathit{s\_until}(\mathit{not}\ END\ \mathit{and}\ AWREADY\ \mathit{and}\ f_{LTL\rightarrow SVA}(F(END)), f_{LTL\rightarrow SVA}(\neg END))\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — $(F \equiv \mathit{s\_eventually})$

7. $\mathit{s\_until}(\mathit{not}\ END\ \mathit{and}\ AWREADY\ \mathit{and}\ \mathit{s\_eventually}(f_{LTL\rightarrow SVA}(END)), f_{LTL\rightarrow SVA}(\neg END))\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — atomic propositions similarly interpreted

7. $\mathit{s\_until}(\mathit{not}\ END\ \mathit{and}\ AWREADY\ \mathit{and}\ \mathit{s\_eventually}(END), f_{LTL\rightarrow SVA}(\neg END))\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — $(\neg \equiv \mathit{not})$

8. $\mathit{s\_until}(\mathit{not}\ END\ \mathit{and}\ AWREADY\ \mathit{and}\ \mathit{s\_eventually}(END), \mathit{not}\ f_{LTL\rightarrow SVA}(END))\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — atomic propositions similarly interpreted

8. $\mathit{s\_until}(\mathit{not}\ END\ \mathit{and}\ AWREADY\ \mathit{and}\ \mathit{s\_eventually}(END), \mathit{not}\ END)\ \mathit{and}$
   $f_{LTL\rightarrow SVA}(U(AWREADY, AWID))$ — $(U \equiv \mathit{s\_until})$

9. $\mathit{s\_until}(\mathit{not}\ END\ \mathit{and}\ AWREADY\ \mathit{and}\ \mathit{s\_eventually}(END), \mathit{not}\ END)\ \mathit{and}$
   $\mathit{s\_until}(f_{LTL\rightarrow SVA}(AWREADY), f_{LTL\rightarrow SVA}(AWID))$ — atomic propositions similarly interpreted

10. $\mathit{s\_until}(\mathit{not}\ END\ \mathit{and}\ AWREADY\ \mathit{and}\ \mathit{s\_eventually}(END), \mathit{not}\ END)\ \mathit{and}$
    $\mathit{s\_until}(AWREADY, f_{LTL\rightarrow SVA}(AWID))$ — atomic propositions similarly interpreted

11. $\mathit{s\_until}(\mathit{not}\ END\ \mathit{and}\ AWREADY\ \mathit{and}\ \mathit{s\_eventually}(END), \mathit{not}\ END)\ \mathit{and}$
    $\mathit{s\_until}(AWREADY, AWID)$

Example 5.4.2.

1.	$f_{LTL\rightarrow SVA}(U(END, AWVALID \rightarrow$ $X(((AWID \leftrightarrow X^{\text{-}1}AWID) \wedge F(\top)) \vee END)))$	$(U \equiv s\_until)$
2.	$s\_until(f_{LTL\rightarrow SVA}(END), f_{LTL\rightarrow SVA}(AWVALID \rightarrow$ $X(((AWID \leftrightarrow X^{\text{-}1}AWID) \wedge F(\top)) \vee END)))$	atomic propositions similarly interpreted
3.	$s\_until(END, f_{LTL\rightarrow SVA}(AWVALID \rightarrow$ $X(((AWID \leftrightarrow X^{\text{-}1}AWID) \wedge F(\top)) \vee END)))$	$\rightarrow \equiv$ implies
4.	$s\_until(END, f_{LTL\rightarrow SVA}(AWVALID)\ implies$ $f_{LTL\rightarrow SVA}(X(((AWID \leftrightarrow X^{\text{-}1}AWID) \wedge F(\top)) \vee END)))$	atomic propositions similarly interpreted
5.	$s\_until(END, AWVALID\ implies$ $f_{LTL\rightarrow SVA}(X(((AWID \leftrightarrow X^{\text{-}1}AWID) \wedge F(\top)) \vee END)))$	X $\equiv$ nexttime
6.	$s\_until(END, AWVALID\ implies$ $nexttime(f_{LTL\rightarrow SVA}(((AWID \leftrightarrow X^{\text{-}1}AWID) \wedge F(\top)) \vee END)))$	$\vee \equiv$ or
7.	$s\_until(END, AWVALID\ implies$ $nexttime(f_{LTL\rightarrow SVA}(((AWID \leftrightarrow X^{\text{-}1}AWID) \wedge F(\top))) \ or\ f_{LTL\rightarrow SVA}(END)))$	$\wedge \equiv$ and
8.	$s\_until(END, AWVALID\ implies$ $nexttime((f_{LTL\rightarrow SVA}((AWID \leftrightarrow X^{\text{-}1}AWID))\ and\ f_{LTL\rightarrow SVA}(F(\top)))\ or\ f_{LTL\rightarrow SVA}(END)))$	$e \leftrightarrow X^{\text{-}1}e \equiv$ \$stable(e)
9.	$s\_until(END, AWVALID\ implies$ $nexttime((\$stable(AWID)\ and\ f_{LTL\rightarrow SVA}(F(\top)))\ or\ f_{LTL\rightarrow SVA}(END)))$	$(F \equiv s\_eventually)$
9.	$s\_until(END, AWVALID\ implies$ $nexttime((\$stable(AWID)\ and\ s\_eventually(f_{LTL\rightarrow SVA}(\top)))\ or\ f_{LTL\rightarrow SVA}(END)))$	$(\top \equiv true)$
10.	$s\_until(END, AWVALID\ implies$ $nexttime((\$stable(AWID)\ and\ s\_eventually(true))\ or\ f_{LTL\rightarrow SVA}(END)))$	atomic propositions similarly interpreted
11.	$s\_until(END, AWVALID\ implies$ $nexttime((\$stable(AWID)\ and\ s\_eventually(true))\ or\ END))$	

5.4 Direct Transformations

This section shows an alternative approach for generating SVA directly from TPL^+ without the necessity of translating TPL^+ to LTL first as we do in the previous section. This section also shows a comparison between the indirect and direct approaches in term of generating SVA formulas.

The direct approach can map only certain commonly-occurring TPL^+-formulas into SVA, unlike the indirect approach which can map any TPL^+ formula into SVA. In this section, we will show later the differences between the two approaches. Now, let us show how we can express some TPL^+ formulas into SVA formulas in a shorter way. For example, the corresponding meaning of sentence (204) in TPL^+ is as follows:

(206) $\{AWREADY^f\}_<[\![AWID]\!]$.

On the other hand, we can express the above TPL^+ formula in SVA as follows:

(207) $first\_match(AWID\ s\_until\ AWREADY)$.

The TPL^+ formula (206) asserts that, within the temporal context $[s, t]$, there is a unique first occurrence t_0 over which $AWREADY$ starts and the interval between the start point s and the unique point t_0 includes the occurrence of $AWID$ at every interval. In contrast, the SVA-formula (207) asserts that, from the start (s) of the evaluation, $AWID$ holds at all times before the first $AWREADY$. Note that, from the semantics of "first_match", the sequence property ($AWID\ s\_until\ AWREADY$) is true at the first occurrence. Both interpretations describe the same truth-condition of sentence (204). For further detail on the semantics of SVA operators, we refer the reader to Section 2.5.

Note that when we discussed TPL^+ and LTL formulas in Section 5.2, we pointed out that TPL^+ is evaluated over intervals which are characterised by two numerical parameters, start and end points. Therefore, we must have another point to represent the end point in any LTL-based language such as SVA. However, SVA formulas are used in practice within a time window that is specified by the system evaluators. Given that the end point parameter does not occur until the end of the SVA simulation, we may omit writing the end point parameter in any SVA formulas as shown in (207).

Let us take another example to obtain a better idea about how we can write SVA formulas that interpret the same truth-conditions as TPL^{+}. Consider the following sentence:

(208) Awid must be asserted within the first burst.

The semantics of sentence (208) in TPL^{+} and SVA are shown below, respectively:

(209) $\{BURST^{f}\}\langle AWID\rangle\top;$

(210) $first\_match(\ AWID\ within\ BURST\ ).$

The TPL^{+} formula (209) asserts that, within the temporal context $[s,t]$, there is a unique first occurrence over which $BURST$ starts and that interval includes an interval over which $AWID$ is true. By contrast, SVA formula (210) asserts that, the first time $BURST$ becomes true, it has to continue until an $AWID$ event has occurred. Again, from the semantics of "first_match", the sequence "$(AWID\ within\ BURST)$" is true at the first occurrence. Thus, both formulas hold the same truth-condition of sentence (208).

From the above examples, there are certain commonly-occurring TPL^{+}-formulas can be translated into SVA without the need of first translating TPL^{+} to LTL. In fact, these TPL^{+}-formulas have simpler SVA translations than those produced by the indirect approach. Table 5.4.1 gives some examples of simpler and better SVA translations than is achieved by the indirect approach. In Table 5.4.1, we define a procedure $\triangle$ for mapping TPL^{+}-formulas into SVA formulas.

As shown in Table 5.4.1, we take advantage of temporal operators in SVA (described in Table 5.3.1) to build up our direct rules. These temporal operators can help us to express all the possible interpretations of temporal behaviours in SVA (see Vijayaraghavan and Ramanathan, 2006).

(R1)	$(\langle e\rangle\top)^{\Delta} = \mathit{s\_eventually}\ e.$
(R2)	$(\|e\rangle\top)^{\Delta} = e.$
(R3)	$([\![e]\!])^{\Delta} = e[{}^{*}1 : \$].$
(R4)	$(\langle e^{\leftleftarrows}\rangle\top)^{\Delta} = \$\mathit{stable}(e).$
(R5)	$(\|n\rangle_{*}[\![e]\!])^{\Delta} = e[{}^{*}n].$
(R6)	$(\|n\rangle_{*}\langle e\rangle\top)^{\Delta} = e[{}^{*}1 : n].$
(R7)	$([e]_{=}\langle e'\rangle\top)^{\Delta} = e\ \ \text{\|->}\ \mathit{s\_eventually}\ e'.$
(R8)	$([e]_{=}\|e'\rangle\top)^{\Delta} = e\ \ \text{\|->}\ e'.$
(R9)	$([e]_{=}\langle e'^{\leftleftarrows}\rangle\top)^{\Delta} = e\ \ \text{\|->}\ \$\mathit{stable}(e').$
(R10)	$([e]_{>}\langle e'\rangle\top)^{\Delta} = e\ \ \text{\|->}\ \#\#1\ \mathit{s\_eventually}\ e'.$
(R11)	$([e]_{>}\|e'\rangle\top)^{\Delta} = e\ \ \text{\|->}\ \#\#1\ e'.$
(R12)	$([e]_{>}\langle e'^{\leftleftarrows}\rangle\top)^{\Delta} = e\ \ \text{\|->}\ \#\#1\ \$\mathit{stable}(e').$
(R13)	$([e]_{>}\|n\rangle_{*}[\![e]\!])^{\Delta} = e\ \ \text{\|->}\ \#\#1\ e'[{}^{*}n].$
(R14)	$([e]_{>}\|n\rangle_{*}\langle e'\rangle\top)^{\Delta} = e\ \ \text{\|->}\ \#\#1\ e'[{}^{*}1 : n].$
(R15)	$(\{e^{l}\}_{>}\langle e'\rangle\top)^{\Delta} = \mathit{e.ended}\ \ \text{\|->}\ \#\#1\ \mathit{s\_eventually}\ e'.$
(R16)	$(\{e^{l}\}_{>}\|e'\rangle\top)^{\Delta} = \mathit{e.ended}\ \ \text{\|->}\ \#\#1\ e'.$
(R17)	$(\{e^{l}\}_{>}[\![e']\!])^{\Delta} = \mathit{e.ended}\ \ \text{\|->}\ \#\#1\ e'[{}^{*}1 : \$].$
(R18)	$(\{e^{l}\}_{>}\langle e'^{\leftleftarrows}\rangle\top)^{\Delta} = \mathit{e.ended}\ \ \text{\|->}\ \#\#1\ \$\mathit{stable}(e').$
(R19)	$(\{e^{l}\}_{>}\|n\rangle_{*}[\![e]\!])^{\Delta} = \mathit{e.ended}\ \ \text{\|->}\ \#\#1\ e'[{}^{*}n].$
(R20)	$(\{e^{l}\}_{>}\|n\rangle_{*}\langle e'\rangle\top)^{\Delta} = \mathit{e.ended}\ \ \text{\|->}\ \#\#1\ e'[{}^{*}1 : n].$
(R21)	$([e]_{<}\|e'\rangle\top)^{\Delta} = e\ \ \text{\|->}\ \$\mathit{past}(e', 1).$
(R22)	$(\{e^{f}\}_{=}[\![e]\!])^{\Delta} = \mathit{first\_match}(\ e'\ \mathit{throughout}\ e\ ).$
(R23)	$(\{e^{f}\}_{=}\langle e'\rangle\top)^{\Delta} = \mathit{first\_match}(\ e'\ \mathit{within}\ e\ ).$
(R24)	$(\{e^{f}\}_{<}\langle e'\rangle\top)^{\Delta} = \mathit{not}\ e\ \mathit{s\_until}\ (\ e'\ \mathit{and}\ \mathit{not}\ e\ ).$
(R25)	$(\{e^{f}\}_{<}[\![e']\!])^{\Delta} = \mathit{first\_match}(e'\ \mathit{s\_until}\ e).$
(R26)	$(\{e^{f}\}_{<+}[\![e']\!])^{\Delta} = \mathit{first\_match}(e'\ \mathit{s\_until\_with}\ e).$

Table 5.4.1: The direct rules map commonly-occurring TPL^{+}-formulas into simpler SVA.

Moreover, in Table 5.4.1, we use *first_match* and *.ended* operators to give equivalent interpretations for the event-relations *first* and *last*, respectively. For example, if an expression has multiple matches, using the "first_match" operator will consider only the first occurrence of this expression to be true and ignore the rest of the matches. In contrast, if a signal has occurred for several time points, using the ".ended" operator will only be considered if the last occurrence of this expression is true. Therefore, using these operators help us to match up with the semantics of the event-relations *first* and *last* in TPL^+.

Note that all the rules, in Table 5.4.1, are constructed in non-recursive structures. This is because some *SVA* operators require that certain event types be combined with them. For example, the *throughout* operator is used to assert that a certain expression is valid over the period of the sequence. Therefore, in its corresponding TPL^+ formula in rule (R22), the second argument of the modal operator $=$ must be restricted with universal quantification. Another example is that the *within* operator is used to assert that there is the containment of an expression within a sequence. Therefore, in its corresponding TPL^+ formula in rule (R23), the second argument of the modal operator $=$ must be restricted with existential quantification. Thus, we must have the knowledge about the quantification type of the second argument in some TPL^+ formulas in order to provide the equivalent and correct interpretations in *SVA*.

Let us show how we can map TPL^+ formulas (206) and (209) into *SVA*-formulas (207) and (210) using our direct transformation rules in Table 5.4.1. Mapping both TPL^+ formulas into *LTL* can be done as shown in Examples 5.4.3 and 5.4.4, respectively.

Example 5.4.3.
$(\{AWREADY^f\}_<[\![AWID]\!])^\Delta = first\_match(AWID\ s\_until\ AWREADY)$ – (R25)

Example 5.4.4.
$(\{BURST^f\}\langle AWID\rangle\top)^\Delta \ = \ first\_match(\ AWID\ within\ BURST\ )$ – (R23)

By applying our direct rules, we successfully mapped TPL^+ formulas (206) and (209) into SVA formulas (207) and (210).

Now, let us show the differences between direct and indirect approaches for mapping TPL^+ into SVA. Figure 5.4.1 shows a sketch of our model for generating SVA from English sentences and how both approaches have been used in our model.

$$\begin{tikzcd}
\text{TempCNL Sentences} \arrow[r, "f\ _{TempCNL \to TPL^+}"] & TPL^+ \arrow[r, "f_{TPL^+ \to LTL}"] \arrow[d, "\triangle : f_{TPL^+ \to SVA}"'] & LTL \arrow[d, "f_{LTL \to SVA}"] \\
 & SVA \arrow[r, leftrightarrow, "\text{They are equivalent.}"'] & SVA
\end{tikzcd}$$

Figure 5.4.1: Our model for generating SVA from TempCNL sentences

As shown in Figure 5.4.1, the indirect approach is started by mapping TPL^+ into LTL and then translating LTL formulas into SVA. We know that the indirect approach is a valid method based on our proof of Theorem 5.2.1 which states that any TPL^+ formula can be mapped into an LTL formula. (Recall that if ψ is a TPL^+ sentence, then the corresponding LTL sentence is given by $\#(\psi, END)$ where END represents the end point in TPL^+.) Moreover, we also observe that any LTL formula can be expressed in SVA after IEEE Std 1800-2009 (2009) was published which included all LTL operators, as discussed in Section 5.3.

In the direct approach, we map TPL^+ into SVA without the necessity of mapping TPL^+ into LTL first. SVA is a combination of LTL, regular expressions, and some features corresponding to clocks and resets. Thus, SVA has rich temporal operators that allow us to express an statement in various ways. However, the direct rules, in Table 5.4.1, are non-recursive rules. This makes the direct rules more restrictive than the rules in Definition 5.2.1. Therefore, the direct rules can be more efficient with the TPL^+

formulas that have nesting depth of at most 2, unlike the transformations in Section 5.2 which have more recursive rules, which in turn render the transformations quite effective for generating SVA from TPL^+. We would like to have simple translations from TPL^+ to SVA that works generally, but we only have them for fixed cases. This work will be done in the future.

We claim the direct approach is valid for mapping TPL^+ to SVA similar to the indirect one. Therefore, if there is a possible generated SVA formula from the direct approach, it must be semantically equivalent to the generated SVA formula from the indirect approach but they may have different syntactic forms. Thus, the results from both approaches are equivalent in the following sense. We assume that

$$\mathfrak{A} \models_s f_{TPL^+ \to SVA}(\psi),$$

where $\mathfrak{A}$ is a finite model. Let us take $\mathfrak{A}'$ to be the result of including an event END occurring after the last event occurring in e. We claim that

$$\mathfrak{A}' \models_s f_{LTL \to SVA}(f_{TPL^+ \to LTL}(\psi)).$$

Conversely, we take any model of $\mathfrak{A}' \models_s f_{TPL^+ \to LTL}(f_{LTL \to SVA}(\psi))$ to be a model of $\mathfrak{A} \models_s f_{TPL^+ \to SVA}(\psi)$. Therefore, we can generate a SVA formula from any TPL^+ formula using the direct rules in Table 5.4.1 that carry out the same truth-condition of the SVA formula that is generated using transformation rules that listed in Definition 5.2.1 and the LTL operators in SVA (as shown in Table 5.3.1).

Now, let us take two example cases and check if the generated SVA formulas in each case are equivalent. Figure 5.4.2a shows us the first case where we apply the same suggested model in Figure 5.4.1.

In Figure 5.4.2a, we show how we map the TPL^+ formula to the LTL formula in Example 5.2.2. Then, we show how we transform the LTL formula to the SVA formula in Example 5.4.1. For the direct approach, we map the TPL^+ formula to the SVA formula in Example

$$\begin{tikzcd}
\{AWREADY^{f}\}_{<}[\![AWID]\!] \arrow[r, "f_{TPL^{+}\to LTL}"] \arrow[d, "f_{TPL^{+}\to SVA}"'] & U(\neg END \wedge AWREADY \wedge F(END), \neg END)\wedge U(AWREADY, AWID) \arrow[d, "f_{LTL\to SVA}"] \\
first\_match(AWID\ s\_until\ AWREADY) \arrow[r, leftrightarrow, "\text{They are equivalent.}"'] & s\_until(not\ END\ and\ AWREADY\ and\ s\_eventually(END), not\ END)\ and\ s\_until(AWREADY, AWID)
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
{[AWVALID]}_{>}\langle AWID^{\Leftarrow}\rangle\top \arrow[r, "f_{TPL^{+}\to LTL}"] \arrow[d, "f_{TPL^{+}\to SVA}"'] & U(END, AWVALID \to X(((AWID \leftrightarrow X^{-1}AWID) \wedge F(\top)) \vee END)) \arrow[d, "f_{LTL\to SVA}"] \\
AWVALID\ \ |\text{->}\ \#\#1\ \$stable(AWID) \arrow[r, leftrightarrow, "\text{They are equivalent.}"'] & s\_until(END, AWVALID\ implies\ nexttime((\$stable(AWID)\ and\ eventually(true))\ or\ END))
\end{tikzcd}$$

(b)

Figure 5.4.2: Two cases for evaluating the *SVA* formulas in our two approaches.

5.4.3. Let us now compare the *SVA* results and see if they have the same truth-condition. The *SVA* results in Figure 5.4.2a are repeated here as follows:

(211) $s\_until(not\ END\ and\ AWREADY\ and\ s\_eventually(END), not\ END)\ and$
$s\_until(AWREADY, AWID);$

(212) $first\_match(AWID\ s\_until\ AWREADY).$

The *SVA* formula (211) is generated using the indirect approach, and a typical model is shown in Figure 5.4.3. On the other hand, the *SVA* formula (212) is generated using the direct approach, and a typical model is shown in Figure 5.4.4.

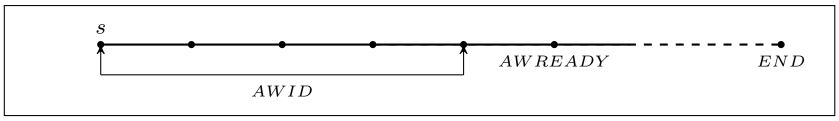


Figure 5.4.3: A model for the *SVA* formula (211).

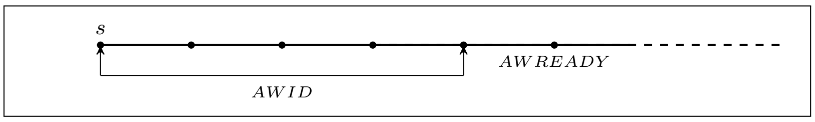


Figure 5.4.4: A model for the *SVA* formula (212).

In Figure 5.4.3, a model in which the *SVA* formula (211) holds is as follows: there is an event *END* occurs somewhere, and some time strictly before that there is a unique first occurrence over which *AWREADY* starts, and at all times before that *AWID* occurs. On the other hand, in Figure 5.4.4, a model in which the *SVA* formula (212) holds is as follows: *AWID* occurs at all times before the first *AWREADY*.

Now, let us compare those models. If we have a model construction such as in Figure 5.4.3 which is a model of the *SVA* formula (211), it will be model of the *SVA* formula (212). Conversely, if we have a model construction such as in Figure 5.4.4 which is a model of the *SVA* formula (212) and we add an event *END* after all other events in the model, it will be model of the *SVA* formula (211).

Figure 5.4.2b shows another example case for evaluating the equivalence relation between the *SVA* results that generated from both approaches. In this figure, we show how we map the TPL^+ formula to the *LTL* formula in Example 5.2.3. Then, we show how we transform the *LTL* formula to the *SVA* formula in Example 5.4.2. For the direct approach, we map the TPL^+ formula to the *SVA* formula using (R12) in Table 5.4.1. Let us now compare the *SVA* results and see if they have the same truth-condition. The *SVA* results in Figure 5.4.2b are repeated here as follows:

(213) $s\_until(END, AWVALID\ implies$
$nexttime((\$stable(AWID)\ and\ eventually(true))\ or\ END));$

(214) $AWVALID\ \ |\text{->}\ \#\#1\ \$stable(AWID).$

The *SVA* formula (213) is generated using the indirect approach, and a typical model is shown in Figure 5.4.5. On the other hand, the *SVA* formula (214) is generated using the direct approach, and a typical model is shown in Figure 5.4.6.

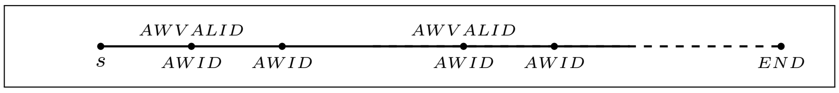


Figure 5.4.5: A model for the *SVA* formula (213).

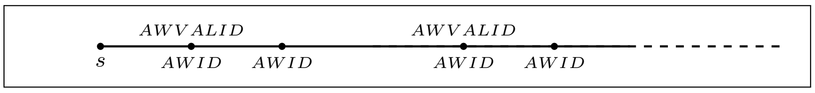


Figure 5.4.6: A model for the *SVA* formula (214).

In Figure 5.4.5, a model in which the *SVA* formula (213) holds is as follows: there is an event *END* occurs, and before that every time that *AWVALID* occurs must immediately be followed by a time point which includes the occurrence of *AWID* if and only if *AWID* held at the previous time point. On the other hand, in Figure 5.4.6, a model in which the *SVA* formula (214) holds is as follows: immediately after every time over which *AWVALID* occurs, there exists a time point over which *AWID* holds if and only if *Awvalid* held at the previous time point.

The proof of this case is very similar to the previous case. If we have a model construction such as in Figure 5.4.5 which is a model of the *SVA* formula (213), it will be model of the *SVA* formula (214). Conversely, if we have a model construction such as in Figure

5.4.6 which is a model of the *SVA* formula (214) and we add an event END after all other events in the model, it will be model of the *SVA* formula (213).

The procedure in all other (twenty-four) cases is similar. We take a TPL^+ formula ψ of each of the forms on the left-hand side of Table 5.4.1 (i.e. the argument of the function Δ). We compute the translation $\#(\psi, END)$, and then map to *SVA* using the translation in Table 5.3.1. We get an *SVA* formula ξ. We then show that any (finite) model of ξ is a model of ψ^Δ, and that any finite model of ψ^Δ, together with an extra END-event appended at the end, is a model of ξ. The checking (laborious, but routine) is similar in character to the cases (R25) and (R12) considered above. Of course, both ξ and ψ^Δ will each have infinitely many models; however, these will in all cases always have a simple general form very much as that depicted in Figures 5.4.3 – 5.4.6 for the cases (R25) and (R12). In this way, the required equivalences can be checked in a similar way. Thus, the translations given in Δ amount to nothing more than a series of handy *SVA* short-cuts for commonly occurring forms of formulas.

Based on our comparison of direct and indirect approaches, the generated *SVA* formulas from TPL^+ using both transformations are equivalent. However, the indirect approach is more general than the direct approach as mentioned previously. Note that both transformation approaches will be integrated with the TPE system (discussed in Section 4.1) to make generating *LTL* and *SVA* from TPL^+ possible in practice. Thus, we will call the TPE system that uses direct approach D-TPE to distinguish it from the TPE system with the indirect approach.

5.5 Conclusion

In this chapter, we began with an overview of the expressiveness of *FOL* and *LTL*. Next, we showed our transformation rules for generating *LTL* from TPL^+. We proved that mapping TPL^+ into *LTL* is possible using our transformation rules. Then, we showed the mapping steps of generating LTL from TPL^+ using our rules. Next, we presented the translations from LTL to SVA with a practical example of the transformations. Finally, we showed an alternative approach for generating SVA directly from TPL^+ using our direct transformations. Again, we demonstrated the mapping procedures by giving some examples. In the following chapter, we undertake experimental work to prove in practice the relative validity of each approach.

Chapter 6

Evaluation

In this chapter, we begin by describing the dataset collections and our efforts at refining them for evaluating the D-TPE and TPE systems. Next, we present several experiments in generating TPL^+ and SVA from natural language specifications. Section 6.2 describes an evaluation of the TPE's performance for generating TPL^+ and SVA. Section 6.3 illustrates a comparison between the TPE system and an existing tool for capturing SVA from natural language descriptions. Section 6.4 makes a comparison between the D-TPE and TPE systems for generating SVA.

6.1 Dataset Collections

There are few resources available for specifying system requirements in SVA. There are two possible reasons for the limitation of SVA resources: (i) SystemVerilog assertion became more effective after adding major new features including LTL operators in IEEE Std 1800-2009 (2009), and (ii) some computer hardware companies do not make their source code available to the public. Thus, we have only collected two datasets for testing the TPE and D-TPE systems: (1) The Advanced Microcontroller Bus Architecture (AMBA) (see ARM Ltd, 2009, 2012a,b), and (2) Open Core Protocol (OCP) (see OCP-IP Association, 2013). We begin with a general description of our datasets.

6.1.1 AMBA dataset

The AMBA specification defines an on-chip communications standard for designing high-performance embedded micro-controllers. The AMBA specifications, developed by ARM Holdings, help to simplify developing multi-processor designs that often have large numbers of controllers and peripherals. Nowadays, AMBA is widely used on a range of Application-Specific Integrated Circuit (ASIC) and System-on-Chip (SoC) parts having applications processors employed in portable mobile devices such as smartphones and tablets. The AMBA dataset contains 396 *SVA*s specifying system requirements together with English comments explaining their meanings. However, these 396 *SVA*s are written without the inclusion of *LTL* operators. Therefore, we manually shifted them into *SVA* with *LTL* operators to allow us to compare them with the produced *SVA*s. In fact, since we will generate *SVA* formulas using two different systems, we must shift all original *SVA*s into two different formats that are suitable for both systems. For example, sentence (215) and its *SVA* interpretation in (216) are taken from the AMBA dataset. Sentence (215) is acknowledged as a TempCNL sentence which can be processed by our parser as presented in Section 4.1. In view of its interpretation, it states that every point must include the occurrence of *Awvalid* until the time point over which *Awready* is high.

(215) Awvalid must remain asserted until Awready goes high.

(216) $AWVALID$ & $!AWREADY$ ##1 |–> $AWVALID$.

Because *SVA* formula (216) was written without the inclusion of the *s_until* operator, *SVA* formula (216) is difficult to extract from TPL^+ using the TPE and D-TPE systems. On other hand, *SVA* formulas (217) and (218), for example, are more convenient than (216); especially from practical viewpoints for mapping TPL^+ into *SVA* using the TPE and D-TPE systems.

(217) $s\_until(not\ END\ and\ AWREADY\ and\ s\_eventually(END), not\ END)\ and$
$s\_until(AWREADY\ and\ s\_eventually(END), AWVALID).$

(218) $first\_match(AWVALID\ s\_until\ AWREADY).$

Thus, we prefer formulas (217) and (218) since there is no available theorem proving system that can check the equivalence between those *SVA* forms. Moreover, most of such *SVA* formulas were written before including *LTL* operators (see IEEE Std 1800-2009 (2009)). Note that *SVA* formulas (217) and (218) are equivalent to statement (215) based on the proof that the generated *SVA* formulas from TempCNL sentences using the TPE and D-TPE systems are equivalent (as shown in Section 5.4). Henceforth, to make a clear distinction of two different formats such as (217) and (218), we call the AMBA dataset that is suitable for the TPE system *SVA_AMBA*, and we call the AMBA dataset that is suitable for the D-TPE system *DSVA_AMBA*.

6.1.2 OCP dataset

The OCP specification defines a high-performance, bus-independent interface between IP cores. The OCP specifications, developed by the OCP Working Group, help to reduce design risk, design time, and manufacturing costs for System-on-Chip designs. The OCP dataset contains 432 English comments explaining system requirements and does not contain any of their meanings in *SVA*. Therefore, for each of 432 English comments, we manually wrote two different formats of SVA, but they are semantically equivalent; similar to the process we employed with the AMBA dataset. Hence, we call the OCP dataset that is suitable for the TPE system *SVA_OCP*, and we call the OCP dataset that is suitable for the D-TPE system *DSVA_OCP*.

Unfortunately, those SVAs are written in an abstract way, since we do not know the word-knowledge for most of the noun phrases. For example, consider the following:

(219) MTagID must remain high during every non inorder request phase.

where the noun phrase"the non inorder request phase" refers to a specific term in the system design that is difficult to extract without the source codes of the OCP specification. Therefore, we leave it to the designers to decide on the suitable interpretation, as shown in (220).

(220) $s\_until(END, [\mathbf{non\_inorder}(\mathbf{request\_phase})]\ and$
$s\_until(not\ [\mathbf{non\_inorder}(\mathbf{request\_phase})], MTAGID))$.

The expression, highlighted in bold font, must be resolved by the OCP designers to have the full meaning in SVA. Otherwise, we will consider the generated SVA formula to be false since it has one argument that requires its *SVA*'s term. We shall call this problem the "lack of world-knowledge". Despite this, for AMBA and OCP datasets, both TPE and D-TPE systems have approximately 814 words with their logical forms in *SVA*.

6.1.3 Differences between AMBA and OCP Datasets

In this section, we analyse both datasets in terms of (i) the number of temporal prepositions that occur per sentence, (ii) the number of pronominal anaphors, and (iii) the average length of sentence and maximum length of sentence. Showing this analysis helps us estimate the challenges we may face when we run our experiments.

Let us show first the number of temporal prepositions that occur per sentence as presented in Table 6.1.1.

Dataset	Sentences	No TP	One TP	Two TPs	Three TPs	Four TPs
AMBA	396	82	280	32	2	n/p
OCP	432	132	166	110	21	3

Table 6.1.1: The number of temporal prepositions that occur per sentence.

As shown in Table 6.1.1, the most frequent number of temporal prepositions that occur in both datasets is one per sentence. In contrast, the least frequent number of temporal prepositions that occurs in the AMBA dataset is 3 per sentence, while the least frequent number of temporal prepositions that occurs in the OCP dataset is 4 per sentence. We notice that there are a relatively large number of sentences in both datasets that do not have a temporal preposition. Thus, the performance of the TPE and D-TPE systems will highly depend on how much knowledge we have inserted into these systems for non-temporal sentences. These analyses show us expected challenges such as capturing complex prepositions in TPL^{+} and mapping them into SVA.

Let us turn to the analysis of the number of pronominal anaphors, the average sentence length, and maximum sentence length in both datasets. Table 6.1.2 shows the above aspects in both datasets.

Dataset	Pronominal anaphors		Average sentence length	Maximum sentence length
	Personal pronoun	Other types		
AMBA	14	4	12.45	61
OCP	25	3	17.84	78

Table 6.1.2: Statistical Information on Anaphora and sentence length in the both datasets.

As noted in Table 6.1.2, we analyse the number of pronominal anaphors in both datasets based on two different types (personal pronoun and all other types of pronoun). We show this classification on anaphora types because our adopted anaphora resolution can only resolve personal pronouns and lexical anaphors and, recognises expletive pronouns as discussed in Section 4.2. In Table 6.1.2, we find that the occurrence number of personal pronouns in the AMBA dataset is 14 times; whereas it in the OCP dataset is 25 times. Moreover, we found also that the occurrence number of other types of pronoun in the AMBA dataset is 4 times; whereas in the OCP dataset it is 3 times. Here are examples of pronominal anaphors that can be found in our datasets:

(221) When Rvalid is asserted then it must remain asserted until Rready is high.

(222) If either master or slave have connection set to 0 , then ConnectCap for master and slave that have this parameter to 1 must be tied off to 0.

where in sentence (221) the pronoun *it* is a personal pronoun, while in sentence (222) the pronoun *this* is a demonstrative pronoun which belongs to other types of pronoun.

The average length of sentence in the AMBA dataset is 12.45 words per sentence where some sentences contain up to 61 words. On the other hand, the average length of sentence in the OCP dataset is 17.84 words per sentence, where some sentences contain up to 78 words. We mention these facts for showing that long sentences are often hard to parse using the TPE parser.

6.2 Evaluating the TPE System for Generating *SVA*

In this Section, we evaluate the TPE system for generating *SVA*. This evaluation gives us an idea of the performance of the TPE system in specification requirements written in English. Section 6.2.1 shows our experimental method for generating *SVA* with *LTL* operators. In Section 6.2.2, we show our experimental results on the collective datasets.

6.2.1 Experimental Method

This section describes our experimental method for generating *SVA* with *LTL* operators from natural language specifications using the TPE system. We set up our experimental method as follows:

1. Parsing the English comments to JavaRAP tool to check if any anaphora occurs or not. Then, we replace the anaphor with its antecedent as explained in Section 4.2.

2. Translating the given comments to TPL^+ formulas using the TPE parser if these comments, of course, have the same TempCNL syntax.

3. Generating *LTL* from TPL^+ formulas using our transformation rules (described in Section 5.2).

4. Translating *LTL* formulas into *SVA* formulas using the equivalent operators in Table 5.3.1.

5. Comparing *SVA* formulas that are generated from the AMBA dataset with *SVA_AMBA*, and comparing *SVA* formulas that are generated from the OCP dataset with *SVA_OCP* as discussed in Section 6.1.

To illustrate our experimental method, Figure 6.2.1 shows the process of evaluating the TPE system based on our dataset collections in Section 6.1.

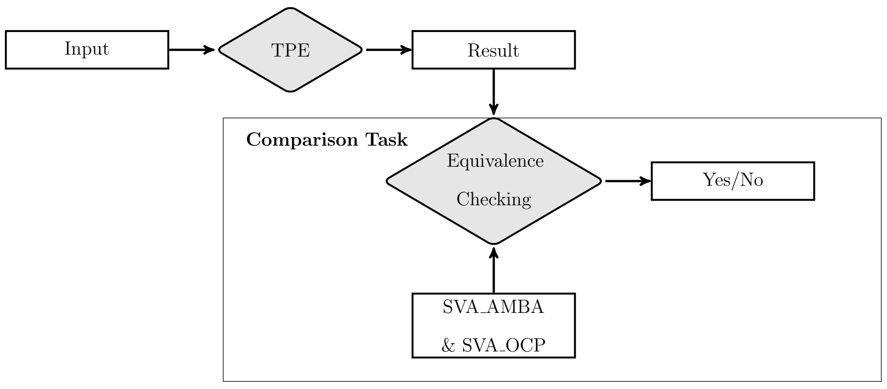


Figure 6.2.1: The steps in our experimental method.

As shown in Figure 6.2.1, we will check the consistency between the generated *SVA* formulas and the corresponding meanings in *SVA_AMBA* and *SVA_OCP*. After we run all sentences using the TPE system, we will sum up all the successful results and compute the accuracy of the TPE system for generating *SVA*.

Besides showing TPE's performance on generating *SVA*, we will show the performance of the JavaRAP tool and the TPE parser. These components can have a significant impact on TPE's performance as we see in the next section.

6.2.2 Results

In this section, we discuss our experimental results from generating *SVA* from AMBA and OCP datasets using the TPE system. Our experimental results include the accuracies of resolving anaphora, parsing sentences for extracting TPL^+, and generating *SVA* semantics. We measure their accuracies as follows:

- To obtain the accuracy of resolving anaphora using the JavaRAP tool, we will check how many cases, including pronominal anaphors, can be resolved. For example sentence (221) contributes to success in JavaRAP, whereas sentence (222) contributes to failure in JavaRAP, since the personal pronoun *it* can be resolved while the demonstrative pronoun *this* can not be resolved.

- To achieve accuracy in parsing sentences using our context-free grammar, we will check how many cases, excluding ungrammatical sentences, can be parsed. For example, sentence (119), which is repeated here as sentence (223), is counted as a failure in parsing.

  (223) The order in which addresses and the first write data item are produced must match.

  Since TempCNL does not include relative clauses as discussed in Section 4.1, their structure is not defined in our grammar rules. Thus, sentence (223) can not be parsed. However, there are some instances found in our datasets which could be considered ungrammatical. Therefore, we do not count these instances as failure cases in our parser. Consider, for example, sentence (169) which will be repeated here as follows:

  (224) FAIL response must occur on a WRC request.

Even though sentence (224) is a grammatically correct. However, we are not handling the use of indefinites with universal sense. Accordingly, sentence (224) is rejected by our grammar rules. For further clarification, see sentence (169) in Section 4.5.3.

Table 6.2.1 shows the experimental results where the performance of the TPE system on the AMBA dataset are better than on the OCP dataset.

Dataset	Sentences	Accuracy		
		Anaphora	Parsing	Semantics
AMBA	396	98.9%	89.3%	81.7%
OCP	432	99.3%	83.2%	59.6%

Table 6.2.1: The performance of the TPE system for generating *SVA*.

We will discuss the above table by examining the results for each column as follows:

- The accuracy of resolving anaphora on the OCP dataset are better than on the AMBA dataset because the JavaRAP tool failed to resolve 4 pronominal anaphors in the AMBA dataset and 3 pronominal anaphors in the OCP dataset. All of these pronominal anaphors are not personal pronouns. These failures are due to the JavaRAP tool since it only resolves third person pronouns.

- In Table 6.2.1, our parsing scores match those achieved by state-of-the-art parsers such as SyntaxNet (Andor et al. (2016)). Here is some justification for these scores comparing with the scores of SyntaxNet parser:

  – Both AMBA and OCP texts are much simpler than the kinds of text being analysed when SyntaxNet parser is evaluated. Generally, our parser gives fairly reasonable results in the context of natural language specifications. However, our parser uses a set of restrictive *CFG* grammars (introduced in Section 4.5). Moreover, these grammar rules were built using only two corpora that are taken from natural language specifications. Thus, if we want to evaluate new corpora, we may need to extend our rules to process these corpora.

- On the other hand, SyntaxNet parser uses a large deep-learning neural network. Therefore, it can provide syntactic analyses for arbitrary English sentences. For example, Andor et al. (2016) evaluates SyntaxNet parser on the English Wall Street Journal (WSJ). Thus, WSJ corpus is a collection of articles written in natural English which our parser, of course, will fail to parse most of its cases.

- Parsing a sentence on the AMBA dataset is more accurate than on the OCP dataset because:

  - The average length sentence in the OCP dataset is 17.84 which is more than the average length sentence in the AMBA dataset (as shown in Table 6.1.2). Generally speaking when English sentences are long, they cause some problems for most of those parsers that are based on context-free grammars as mentioned in Li et al. (1990).

  - As mentioned previously, our context-free grammar does not include relative clauses. Therefore, the number of sentences in the OCP dataset that have relative clauses is larger than the number of sentences in the AMBA dataset. For this reason, the TPE's parser preforms better with sentences in the AMBA dataset.

- Generating *SVA* semantics on the AMBA dataset is more accurate than on the OCP dataset because in the AMBA dataset the number of sentences that exhibit lack of word knowledge is less than the number of sentences in the OCP dataset. These sentences exhibit lack of word knowledge for the following reasons:

  - ARM Holdings has provided SVAs together with their English comments. Therefore, we successfully encode most of them with the TPE system. In contrast the OCP Working Group has only provided English comments explaining OCP system requirements. Therefore, we can not encode most of the technical expressions with the TPE system.

– Another observation is that it can be difficult to extract the corresponding semantics in *SVA* of predicates with more than one argument. In our experiments, we notice that the AMBA dataset has more one-place predicate numbers than the OCP dataset.

The above results show that the TPE system is sufficiently capable of (i) capturing temporal expressions in TPL^+ from natural language specifications, and (ii) generating *SVA* from TPL^+. Note that, in Section 6.4, we compare the performance results of the TPE and D-TPE systems. Moreover, Chapter 7 offers some suggestions for improving the TPE's performance.

6.3 Comparing the TPE with an Existing Tool

In this section, we will compare the accuracy of the TPE system with an existing tool developed in Harris (2013). This tool comprises the only approach intended to capture *SVA* from natural language descriptions. The tool is constructed on an attribute grammar approach that is introduced in Engelfriet (1984). An attribute grammar is a context-free grammar with a set of attributes, assignment of attribute values, evaluation rules, and conditions.

The dataset used in Harris (2013)'s tool evaluation was taken from the AMBA 3 AXI Protocol Checker in ARM Ltd (2009). The AMBA 3 AXI dataset only includes 117 sentences with their equivalent *SVA*s. Therefore, we compare the TPE with the Harris (2013)'s tool using the same benchmark set. As a result, *SVA* success rates differ significantly between ours and Harris (2013)'s tool as shown in Table 6.3.1.

Tool	*SVA* success rate
TPE	69.2%
Harris (2013)'s Toolkit	44%

Table 6.3.1: Comparison between the TPE system and Harris (2013)'s Toolkit.

The reason why a more accurate result occurs in the TPE system than in Harris (2013)'s tool for generating *SVA* from natural language descriptions is that the TPE system captures all temporal expressions including temporal prepositions in TPL^{+} which is expressive enough for dealing with all the complex temporal behaviours that are described in requirements specifications.

In contrast, Harris (2013)'s approach is intended to capture *SVA* formulas directly from natural language descriptions which is potentially risky due to the lack of deep semantic interpretations of sentences featuring temporal information. As we observed in Harris (2013)'s approach, the logic base was Boolean algebra which employs very basic operations such as implication, conjunction, disjunction, and negation. For example, the key words "if", "then", and "when" are expressed using an implication operation. Even though we might agree that the meaning of *when* does not particularly express temporality, we do not know how other temporal prepositions can be expressed using these basic operations.

In summary, Harris (2013)'s approach fails to handle the issues of temporal constructions in natural language descriptions. This approach generally offers a suitable way to generate *SVA* from natural language requirements. On the other hand, our approach aims to be more focused on specifying temporal expressions in natural language assertion descriptions and provides an efficient method for generating *SVA* involving temporal behaviours.

6.4 Comparing the TPE and D-TPE Systems

In this section, we compare the performance of TPE and D-TPE systems for generating *SVA*. The motivation is to compare the efficiency of both systems in generating *SVA* from TPL^+.

We present the performance results of the TPE system for generating *SVA* in Section 6.2.2. Therefore, we only need to obtain the performance results of the D-TPE system and compare it with the performance results of the TPE system. Our evaluation steps are as follows: (1) we generate *SVA* formulas from TPL^+ using D-TPE; (2) we compare the generated *SVA* formulas with *DSVA_AMBA* and *DSVA_OCP* where we consider any generated *SVA* is true if it is equivalent to the corresponding meaning in either *DSVA_AMBA* or *DSVA_OCP*; and (3) finally we compare the accuracy of the D-TPE and TPE systems for generating *SVA*.

Here is an example where we contribute to success in both systems:

(225) Awvalid is low for one cycle after Arestn goes high.

where we can map it into *SVA* using the D-TPE and TPE systems as shown, respectively:

(226) $s\_until(AREASTN\ and\ s\_until(not\ AREASTN\ and$
$not\ AWVALID\ and\ s\_eventually\ END\ and$
$s\_until(END, not\ AREASTN), not\ END\ and\ AREASTN), not\ END)$

(227) $AREASTN.ended\ \ |\!\!-\!\!>\ \#\#1\ not\ AWVALID[{}^*1].$

Both formulas assert that, immediately after the last occurrence of *AREASTN*, *AWVALID* must be low for one cycle. Note that in this evaluation we do not compare the outputs of both systems with each other, but instead we compare the accuracy of both systems for generating *SVA*. We refer the reader to Section 5.4 for further details regarding mapping procedure from TPL^+ into *SVA* using both systems.

Let us see an example where we contribute to the failure of the D-TPE system and to the success of the TPE system:

(228) MRespAccept must be invalid for every non BLCK request during the request phase.

where the corresponding meaning in TPL^+ is:

(229) $\{REQUEST\text{-}PHASE^f\}_=[NON\text{-}BLCK\text{-}REQUEST]_=[\neg MRESPACCEPT]$

We can map TPL^+ formula (229) into SVA using the indirect approach as shown in (230).

(230) $s\_until((MCMD\ and\ MADDR\ and\ MDATA)\ and$
$s\_until(not\ (MCMD\ and\ MADDR\ and\ MDATA),\ BURSTSEQ\_BLCK\_ENABLE\ and$
$s\_until(not\ BURSTSEQ\_BLCK\_ENABLE, not\ MRESPACCEPT))\ and$
$s\_until(not\ (MCMD\ and\ MADDR\ and\ MDATA), not\ END), not\ END)\ and$
$s\_until(END, not\ (MCMD\ and\ MADDR\ and\ MDATA))$

where "$not\ BURSTSEQ\_BLCK\_ENABLE$" belongs to "$NON\text{-}BLCK\text{-}REQUEST$" and "$(MCMD\ and\ MADDR\ and\ MDATA)$" belongs to "$REQUEST\text{-}PHASE$". The nesting depth of TPL^+ formula (230) is 2. However, this TPL^+ formula can not be mapped using the direct rules in Table 5.4.1. Thus, the D-TPE system fails to map this sort of TPL^+ formulas into SVA.

The performance results of the TPE and D-TPE systems for generating SVA from English comments are summarised in Table 6.4.1.

Dataset	Sentences	TPE	D-TPE
AMBA	396	81.7%	81.7%
OCP	432	59.6%	51.5%

Table 6.4.1: The performance results of both systems for generating SVA.

As shown in Table 6.4.1, surprisingly both systems perform the same thing for generating SVA from sentences in the AMBA dataset, because: (1) the AMBA dataset has few cases where sentences have two or more temporal prepositions, and these cases either fail to be parsed or have lack of word-knowledge issues; and (2) the successful cases have used the available non-recursive rules in the D-TPE system at Table 5.4.1. Thus, the number of generated SVAs using the TPE system is the same number of sentences when we use the D-TPE system.

For the OCP dataset, Table 6.4.1 shows that the TPE system performs better than the D-TPE system, because there are 29 TPL^+ formulas, which have a nesting depth of 2 temporal operators which can not cope with the non-restrictive rules in the D-TPE system. In the end, the TPE system is more efficient than the D-TPE system, because the TPE system has recursive rules while the D-TPE system has no recursive rules.

6.5 Conclusion

In this chapter we perform several experiments for generating TPL^+ and SVA with and without LTL operators from natural language specifications. The first experiment is on generating TPL^+ and SVA from AMBA and OCP datasets using the TPE system. We have learned from this experiment that generating SVA from TPL^+ gives outstanding results. The second experiment compares the accuracy of TPE system with an existing tool. From this experiment we deduce that capturing SVA formulas directly from natural language descriptions gives a lower chance for achieving an efficient result, since natural languages such as English require more expressive power than SVA for expressing the semantics of natural language constructions. The last experiment compares the performance results between the TPE and D-TPE systems for generating SVA from English comments. We deduce from the last experiment that the TPE system is better than the D-TPE system. In the next chapter, we suggest some ways for improving the TPE's accuracy, since it has the potential to become one of the most promising systems for generating SVA from natural language specifications.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis aspires to design a temporal controlled natural language for extracting formal specifications, namely *SVA*. In chapter 3, we show several approaches that attempt to provide formal representations for temporal expressions. From those approaches, we have chosen the *TPL* language (discussed in Section 3.3) as a logical form for the TPE system. *TPL* is expressive enough to capture temporal expressions (such as temporal prepositions) in English.

In Chapter 4, we show how the TPE system is constructed. The TPE system consists of an anaphora resolution, TempCNL lexicons and a semantic parser. In the same chapter, we extend *TPL* to handle more temporal expressions in English. This extension was established after we had observed that many temporal constructions were not covered by TPL in Pratt-Hartmann (2005), and those temporal constructions occur commonly in English (such as *within*, *throughout*, *until after*, *since*, *etc*). We then showed how our context-free grammar was defined by combining typed logic with lambda abstraction to obtain TPL^+ formulas. Finally, we restrict some TPL^+ formulas defined in Section 4.4 to simplify their mapping into *LTL* and *SVA*.

In Chapter 5, we present two approaches for generating *SVA* from TPL^+. The first approach is based on indirect transformations where we map first TPL^+ into *LTL* and then translate *LTL* into *SVA*. In this approach, we proved that TPL^+ and *LTL* are logically equivalent in our transformation rules. Moreover, these transformation rules allows us to obtain *LTL* without an explosion problem, unlike the translation approaches in Kamp (1968), Gabbay et al. (1980), McNaughton and Papert (1971), as discussed in Section 5.1. The second approach is based on direct transformation rules where we map directly TPL^+ into *SVA*. Then, each approach is integrated with the TPE system separately. We name the TPE system with the direct approach D-TPE to distinguish it from the TPE system with the indirect approach.

In Chapter 6, we show the results of several experiments on both systems. In Section 6.2, the experiment was conducted on the TPE's performance for generating TPL^+ and *SVA* from natural language specifications. In Section 6.3, the experiment compared the performance of the TPE system with an existing tool for generating only *SVA* from natural language specifications. Section 6.4 showed an experiment on comparing the performance of the TPE and D-TPE systems for generating *SVA*. These experiments confirm that using the TPE system for generating *SVA* from natural language specifications is strongly recommended.

To summarise, in this thesis we make several contributions to the research field of *controlled natural languages*:

1. We have built a controlled natural language that can capture temporal expressions in English.

2. We have extended TPL to capture more temporal constructions.

3. We have constructed a context-free grammar that is annotated with semantic rules for extracting TPL^+ formulas.

4. We have constructed transformation rules to generate *LTL* from TPL^+ without an

explosion problem.

5. We have constructed transformation rules to generate SVA from TPL^+ directly.

6. We have improved the accuracy rate of generating SVA from natural language specifications as shown in Table 6.3.1.

7.2 Future Work

How can the current research be improved? There are many ways to increase the TPE's performance and the efficacy of generated SVA. We give a number of possible suggestions for future work:

1. The TPE system takes only one sentence at a time. Thus, it is worth trying to extend grammar rules to accept more than one sentence. This extension will make the TPE system robust enough to handle the temporal relations that hold between events among multiple sentences.

2. The TPE parser is constructed as a depth-first top-down parser as indicated in Section 4.5. From the limitations of a depth-first top-down parser in Section 2.2, we mentioned that this type takes a long time for parsing long sentences since it requires backtracking and redo operations to produce the complete parse trees. Therefore, this parser is inefficient for long sentences. In this issue, we would investigate further how to improve TPE's performance for parsing long sentences. There are other types of parsers have been developed as indicated in Section 2.2. They might provide more efficient performance than the depth-first top-down parser for parsing long sentences. Thus, further research is required to decide which parsing technique is considered the most efficient one for this problem.

3. The adopted anaphora resolution in the TPE system has a good accuracy for third person pronouns. However, its performance in other anaphora types is less efficient. Therefore, we would investigate further how to reduce the number of sentences rejected by the TPE system due to unresolved anaphora. The possible suggestion for solving this issue is to combine more than one anaphora resolution tool to give better results than each tool in isolation for resolving different types of anaphors. Moreover, if we also make further investigations on resolving anaphora across sentences, this could have a significant impact. For example, in the AMBA dataset, we found that some assertions were expressed in two sentences rather than one sentence. Therefore, including this sort of anaphora will scale up the TPE's performance.

4. As indicated in Section 6.2.2, undetermined word-knowledge for some words can have a major impact on the efficacy of generated SVA. Therefore, we suggest that designing a front end interface to allow the designer to add word-knowledge of unknown expressions into our transformation rules. This work can reduce the number of the lack of word-knowledge in Table 6.2.1.

5. Since there is an intention to design a front end interface, attaching a SystemVerilog simulator (such as Cadence in Simulator (2005), Modelsim in Graphics (2015), and VCS in Synopsys (2004)) with the front end interface would be a positive development. This simulator will help system designers to verify the correctness of generated SVA formulas against their system requirements.

6. An interesting future endeavour is to generate other specification language such as *Property Specification Language* (PSL) and *Computation tree logic* (CTL) using our technique, since those languages possess the same challenges of SVA. The only required step is to construct transformation rules between those languages and TPL^+.

Bibliography

Alfred V Aho and Jeffrey D Ullman. *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice-Hall, Inc., 1972.

James F Allen and George Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, 1994.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. *arXiv preprint arXiv:1603.06042*, 2016.

ARM Ltd. *AMBA 3 AXI Protocol Checker User Guide*, 2009.

ARM Ltd. *ARM AMBA 4 ACE and ACE-Lite Protocol Checkers User Guide*, 2012a.

ARM Ltd. *AMBA 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide*, 2012b.

Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, et al. The forspec temporal logic: A new temporal property-specification language. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 296–311. Springer, 2002.

ASD. *AeroSpace and Defence Industries Association of Europe. Simplified Technical English. Specification ASD-STE100, Issue 6*, 2013.

Avaya Inc. *Avaya Style Guide. Issue 1*, 2004.

Emmon Bach. The algebra of events. *Linguistics and Philosophy*, 9(1):5–16, 1986.

Imran Sarwar Bajwa, Mark G Lee, and Behzad Bordbar. SBVR business rules generation from natural language specification. In *AAAI spring symposium: AI for business agility*, pages 2–8, 2011.

Renate Bartsch and Ferenc Kiefer. *The grammar of adverbials: A study in the semantics and syntax of adverbial constructions.* North-Holland Publishing Company, 1976.

Rainer Bäuerle and Arnim von Stechow. Finite and non-finite temporal constructions in german. In *Time, Tense, and Quantifiers: Proceedings of the Stuttgart Conference on the Logic of Tense and Quantification, Max Niemayer Verlag, Teubingen*, pages 375–421, 1980.

David C Bennett. *Spatial and temporal uses of English prepositions: An essay in stratificational semantics*, volume 17 of *Longman linguistics library.* Longman Publishing Group, 1975.

Patrick Blackburn. Tense, temporal reference, and tense logic. *Journal of Semantics*, 11 (1-2):83–101, 1994.

David S Brée and Ruud A Smit. Temporal relations. *Journal of Semantics*, 5(4):345–384, 1986.

Lauri Carlson. Aspect and quantification in tense and aspect. ed. by Philip Tedeschi and Annie Zaenen. *Syntax and Semantics Ann Arbor, Mich.*, 14:31–64, 1981.

David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

Eugene Charniak. A maximum-entropy-inspired parser. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 132–139. Association for Computational Linguistics, 2000.

Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems,*, 8(2):244–263, 1986.

Bernard Comrie. *Aspect: An introduction to the study of verbal aspect and related problems*, volume 2. Cambridge University Press, 1976.

John A Darringer. The application of program verification techniques to hardware verification. In *Papers on Twenty-five years of electronic design automation*, pages 373–379. ACM, 1988.

Donald Davidson. The logical form of action sentences. *Essays on actions and events*, 5: 105–148, 1967.

David R Dowty. *Word meaning and Montague grammar: The semantics of verbs and times in generative semantics and in Montague's PTQ*, volume 7. Springer, 1979.

David Roach Dowty. *Studies in the logic of verb aspect and time reference in English.* Department of Linguistics, University of Texas at Austin, 1972.

Joost Engelfriet. Attribute grammars: Attribute evaluation methods. *Methods and tools for compiler construction*, pages 103–138, 1984.

Hana Filip. *Aspect, eventuality types and nominal reference.* Taylor & Francis, 1999.

Norbert E Fuchs and Rolf Schwitter. Attempto Controlled English (ACE). *arXiv preprint cmp-lg/9603003*, 1996.

Norbert E Fuchs, Uta Schwertel, and Rolf Schwitter. Attempto Controlled English? not just another logic specification language. In *International Workshop on Logic Programming Synthesis and Transformation*, pages 1–20. Springer, 1998.

Adam Funk, Valentin Tablan, Kalina Bontcheva, Hamish Cunningham, Brian Davis, and Siegfried Handschuh. Clone: Controlled language for ontology editing. In *The Semantic Web*, pages 142–155. Springer, 2007.

Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM, 1980.

Dov M Gabbay, Ian Hodkinson, Mark Reynolds, and Marcelo Finger. *Temporal logic: mathematical foundations and computational aspects*, volume 1. Clarendon Press Oxford, 1994.

Sheila Glasbey. Event structure, punctuality, and when. *Natural Language Semantics*, 12 (2):191–211, 2004.

Mentor Graphics. Modelsim simulator, 2015. URL `http://www.mentor.com/products/fpga/model`.

Ralph Grishman and Beth Sundheim. Message Understanding Conference-6: A brief history. In *Proceedings of the 16th Conference on Computational Linguistics - Volume 1*, COLING '96, pages 466–471, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics. doi: 10.3115/992628.992709. URL `http://dx.doi.org/10.3115/992628.992709`.

Claire Grover, Alexander Holt, Ewan Klein, and Marc Moens. Designing a controlled language for interactive model checking. In *Proceedings of the Third International Workshop on Controlled Language Applications*, pages 29–30, 2000.

Joseph Y Halpern and Yoav Shoham. A propositional modal logic of time intervals. *Journal of the ACM (JACM)*, 38(4):935–962, 1991.

Ian G Harris. Capturing assertions from natural language descriptions. In *Natural Language Analysis in Software Engineering (NaturaLiSE), 2013 1st International Workshop on*, pages 17–24. IEEE, 2013.

James Higginbotham. The logic of perceptual reports: An extensional alternative to situation semantics. *The Journal of Philosophy*, 80(2):100–127, 1983.

James Higginbotham. On semantics. *Linguistic Inquiry*, 16(4):547–593, 1985.

Erhard Hinrichs. Temporal anaphora in discourses of English. *Linguistics and Philosophy*, 9(1):63–82, 1986.

Alexander Holt. Formal verification with natural language specifications: guidelines, experiments and lessons so far. *South African Computer Journal*, pages 253–257, 1999.

J Hopcroft and J Ullman. Introduction to the theory of automata, languages and computation, 1979.

Gila Kamhi, Osnat Weissberg, Limor Fix, Ziv Binyamini, and Ze'ev Shtadler. Automatic datapath extraction for efficient usage of HDD. In *Computer Aided Verification*, pages 95–106. Springer, 1997.

Hans Kamp. A theory of truth and semantic representation. *Formal semantics-the essential readings*, pages 189–222, 1981.

Hans W. Kamp. *Tense Logic and the Theory of Linear Order.* PhD thesis, Computer Science Department, University of California at Los Angeles, USA, 1968.

Tobias Kuhn, Loïc Royer, Norbert E Fuchs, and Michael Schroeder. Improving text mining with controlled natural language: A case study for protein interactions. In *International Workshop on Data Integration in the Life Sciences*, pages 66–81. Springer, 2006.

Carl Lamar. Linguistic analysis of natural language engineering requirements. 2009.

F. Landman. *Events and Plurality: The Jerusalem Lectures.* Events and Plurality. Springer Netherlands, 2001. ISBN 9780792365693.

Richard K Larson. Events and modification in nominals. In *Semantics and Linguistic Theory*, volume 8, pages 145–168, 1998.

Winfred P Lehman. *Descriptive Linguistics: An Introduction. New York: Random House.* 1972.

Wei-Chuan Li, Tzusheng Pei, Bing-Huang Lee, and Chuei-Feng Chiou. Parsing long English sentences with pattern rules. In *Proceedings of the 13th conference on Computational linguistics-Volume 3*, pages 410–412. Association for Computational Linguistics, 1990.

John Lyons. Semantics (vols i & ii). *Cambridge CUP*, 1977.

Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

Robert McNaughton and Seymour Papert. *Counter-free automata.* Research monograph. Cambridge, Mass. M.I.T. Press, 1971. ISBN 0-262-13076-9. URL `http://opac.inria.fr/record=b1082248`.

Albert R Meyer. Weak monadic second order theory of succesor is not elementary-recursive. In *Logic Colloquium*, pages 132–154. Springer, 1975.

George J Milne. *Formal specification and verification of digital systems.* McGraw-Hill, Inc., 1993.

Marc Moens. *Tense, aspect and temporal reference.* PhD thesis, The University of Edinburgh, 1987.

Marc Moens and Mark Steedman. Temporal ontology and temporal reference. *Computational Linguistics*, 14(2):15–28, 1988.

Richard Montague. Formal philosophy: Selected papers of Richard Montague. Number 222–247. Yale University Press, New Haven, London, 1974.

Robert C. Moore. Removing Left Recursion from Context-free Grammars. In *Proceedings of the 1st North American Chapter of the ACL*, pages 249–255. Association for Computational Linguistics, 2000.

OCP-IP Association. Open Core Protocol Specification, Release 3.0, 2013. URL `Available:http://www.accellera.org/downloads/standards/ocp/ocp_3.0/`.

Toshiyuki Ogihara. Adverbs of quantification and sequence-of-tense phenomena. In *Proceedings from Semantics and Linguistic Theory IV*, pages 251–267. DMLL Publications, Cornell University, 1994.

Miles Osborne and CK MacNish. Processing natural language software requirement specifications. In *Requirements Engineering, 1996., Proceedings of the Second International Conference on*, pages 229–236. IEEE, 1996.

Gordon J Pace and Michael Rosner. A controlled language for the specification of contracts. In *International Workshop on Controlled Natural Language*, pages 226–245. Springer, 2009.

Terence Parsons. The progressive in English: Events, states and processes. *Linguistics and Philosophy*, 12(2):pp. 213–241, 1989. ISSN 01650157. URL `http://www.jstor.org/stable/25001338`.

Terence Parsons. Events in the semantics of English: A study in subatomic semantics. 1990.

Barbara H Partes. Nominal and temporal anaphora. *Linguistics and Philosophy*, 7(3): 243–286, 1984.

Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

Ian Pratt and Nissim Francez. On the semantics of temporal prepositions and preposition phrases. Technical Report LCL9701, Computer Science Department, University of Manchester, 1997.

Ian Pratt and Nissim Francez. Temporal prepositions and temporal generalized quantifiers. *Linguistics and Philosophy*, 24(2):187–222, 2001.

Ian Edwin Pratt and David Brée. *The expressive power of the English temporal preposition system.* University of Manchester. Department of Computer Science, 1993.

Ian Pratt-Hartmann. Temporal prepositions and their logic. *Artificial Intelligence*, 166 (1):1–36, 2005.

Arthur N. Prior. *Past, Present and Future.* Oxford University Press, 1967.

Long Qiu, Min-Yen Kan, and Tat-Seng Chua. A public reference implementation of the rap anaphora resolution algorithm. *arXiv preprint cs/0406031*, 2004.

Hans Reichenbach. The tenses of verb. *Elements of Symbolic Logic*, pages 287–298, 1947.

Barry Richards, Inge Bethke, Jaap van der Does, and Jon Oberlander. *Temporal Representation and Inference.* Academic Press, London, 1989.

Ronald G Ross. *Principles of the business rule approach.* Addison-Wesley Professional, 2003.

Görel Sandström. *When-clauses and the temporal interpretation of narrative discourse*, volume 34. Department of General Linguistics, University of Umeå, 1993.

Barry Schein. *Plurals and events*, volume 23 of *ACL-MIT Press Series in Natural Language Processing.* MIT Press, 1993.

Jason G Schlachter. ProNTo Morph: Morphological analysis tool for use with pronto (prolog natural language toolkit). Technical report, University of Georgia, 2003. URL `http://www.ai.uga.edu/mc/pronto`.

Marcel Paul Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8(2):190–194, 1965.

Rolf Schwitter. English as a formal specification language. In *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*, pages 228–232. IEEE, 2002.

Virtuoso Spectre Circuit Simulator. Cadence Design Systems. *Inc., Available at: www. cadence. com*, 2005.

Carlota S Smith. The parameter of aspect (studies in linguistics and philosophy, 43), 1991.

John F Sowa. Common Logic Controlled English. *Draft. URL http://www. jfsowa. com/-clce/specs. htm*, 2004.

IEEE Std 1800-2005. IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. pages 1–648, 2005. doi: 10.1109/IEEESTD.2005.97972.

IEEE Std 1800-2009. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. pages 1–1285, 2009. URL `http://dx.doi.org/10.1109/IEEESTD.2009.5354441`.

IEEE Std 1800-2012. IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. pages 1–1315, Feb 2013. doi: 10.1109/IEEESTD.2013.6469140.

M. J. Steedman. Verbs, time, and modality*. *Cognitive Science*, 1(2):216–234, April 1977. ISSN 1551-6709. URL `http://dx.doi.org/10.1207/s15516709cog0102_4`.

VCS Synopsys. Verilog simulator. *Avaliable HTTP: http://www. synopsys. com/products/simulation/simulation. html*, 2004.

Moshe Y Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences*, 32(2):183–221, 1986.

Zeno Vendler. *Linguistics in Philosophy.* Cornell University Press, 1967.

Henk J Verkuyl. *On the compositional nature of the aspects*, volume 15. Reidel Dordrecht, 1972.

Srikanth Vijayaraghavan and Meyyappan Ramanathan. *A practical guide for SystemVerilog assertions.* Springer, 2006.

Frank Vlach. On situation semantics for perception. *Synthese*, 54(1):129–152, 1983.

Thomas Wilke. Classifying discrete temporal properties. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 32–46. Springer, 1999.

Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1):72–99, 1983.

Adam Wyner, Adeline Nazarenko, and François Lévy. Towards a high-level controlled language for legal sources on the semantic web. In *International Workshop on Controlled Natural Language*, pages 92–101. Springer, 2016.

Appendix A

A list of special terms in Natural Language Specifications

Term	Description	Syntactic Category
Awid	Write address ID.	Proper noun
Awready	Write address ready.	Proper noun
Awvalid	Write address valid.	Proper noun
Acvalid	Valid signal for the snoop address channel.	Proper noun
Acready	Ready signal for the snoop address channel.	Proper noun
Rvalid	Read data valid.	Proper noun
Bvalid	Write response valid.	Proper noun
Arestn	Global Reset Signal.	Proper noun
MTagID	OCP Request tag ID.	Proper noun
MReset	OCP Master reset.	Proper noun
SReset	OCP Slave reset.	Proper noun
MRespAccept	OCP Master accepts response.	Proper noun
Wlast	Write data last transfer indication.	Proper noun
ReadShared	For shareable reads where the master can accept cache line data in any state.	Proper noun

WRC	OCP Write conditional.	Adjective
FAIL	OCP Write conditional fail.	Adjective
EXOKAY	Indicates exclusive access has been successful.	Adjective
Non BLCK	A set of transfers that stays constant throughout the burst.	Adjective
MaxWaits	Maximum number of cycles between two particular events. Its value is 16.	Number
burst	consists of a specified number of transfers.	Temporal noun
response	indicates the status of a transaction.	Temporal noun
transaction	is a variable transfer of data or control between design elements.	Temporal noun
request	is used by a master or slave to access the bus.	Temporal noun
acknowledgement	indicates that a master or slave has completed a read or write transaction.	Temporal noun
assert	The associated boolean variable holds at a particular time.	Intransitive Verb
permit		Intransitive Verb
hold		Intransitive Verb
occur		Intransitive Verb
valid		Adjective
active		Adjective
enabled		Adjective
high		Adjective
de-assert	The associated boolean variable does not hold at a particular time.	Intransitive Verb
disabled		Adjective
illegal		Adjective
absent		Adjective
inactive		Adjective
invalid		Adjective
low		Adjective

Table A.1: A list of terms and their descriptions in natural language specifications

Appendix B

The TPE Grammar

```
/*=====================================================================
                           Inflectional Phrase
=====================================================================*/
ip2([sem:IP]) --> ip1([sem:IP]).
ip2([sem:IP2]) --> ip1([sem:A]), ipcoord([sem:B]), ip1([sem:C]),
                  {var_replace(B,B1),beta(B1@A,S),beta(S@C,IP2)}.
ip2([sem:IP2])--> tpps([sem:TPP]), punctuat,ip1([sem:IP]),
                  {var_replace(TPP,TPP1),beta(TPP1@IP,IP2)}.
ip2([sem:IP2])--> ip1([sem:IP]), tpps([sem:TPP]),
                  {var_replace(TPP,TPP1),beta(TPP1@IP,IP2)}.
ip1([sem:IP2])--> tpp([mclause:M,sem:TPP]), punctuat, ip0([mclause:M,sem:IP]),
                  {var_replace(TPP,TPP1),beta(TPP1@IP,IP2)}.
ip1([sem:IP2])--> ip0([mclause:M,sem:IP]), tpp([mclause:M,sem:TPP]),
                  {var_replace(TPP,TPP1),beta(TPP1@IP,IP2)}.
ip1([sem:IP])--> ip0([mclause:exists,sem:IP]).
ip1([sem:IP])--> ip0([mclause:forall,sem:IP]).
ip0([mclause:M,sem:IP0])--> np1([num:Num,sem:NP]),
                            ibar([num:Num,mclause:M,sem:IBar]),
                            {var_replace(NP,NP1),beta(NP1@IBar,IP0)}.
ip0([mclause:M,sem:IP0])--> np0([num:Num,sem:NP]),
                            ibar([num:Num,mclause:M,sem:IBar]),
                            {var_replace(IBar,IBar1),beta(IBar1@NP,IP0)}.
ip0([mclause:forall,sem:IP0])--> ip([coord:ant,sem:IP1]),
                                 ip([coord:con,sem:IP2]),
                                 {var_replace(IP1,IP11),beta(IP11@IP2,IP0)}.
ip0([mclause:exists,sem:IP0])--> ip([coord:either,sem:IP1]),
                                 ip([coord:or,sem:IP2]),
                                 {var_replace(IP1,IP11),beta(IP11@IP2,IP0)}.
```

```
ip_simple([mclause:M,sem:IP0])--> np1([num:Num,sem:NP]),
                                   ibar([num:Num,mclause:M,sem:IBar]),
                                   {var_replace(NP,NP1),beta(NP1@IBar,IP0)}.
ip_simple([mclause:M,sem:IP0])--> np0([num:Num,sem:NP]),
                                   ibar([num:Num,mclause:M,sem:IBar]),
                                   {var_replace(IBar,IBar1),beta(IBar1@NP,IP0)}.
ip([coord:ant,sem:IP0]) --> if([sem:IF]),punctuat, ip2([sem:IP]),
                            {var_replace(IF,IF1),beta(IF1@IP,IP0)}.
ip([coord:ant,sem:IP0]) --> ip_simple([mclause:_,sem:IP]),if([sem:IF]),
                            {var_replace(IF,IF1),beta(IF1@IP,IP0)}.
ip([coord:con,sem:IP]) --> punctuat, ip2([sem:IP]).
ip([coord:con,sem:IP])--> ip2([sem:IP]).
ip([coord:either,sem:IP0])--> either([sem:EI]), ip2([sem:IP]),
                              {var_replace(EI,EI1),beta(EI1@IP,IP0)}.
ip([coord:or,sem:IP])--> [or], ip2([sem:IP]).
/*==========================================================================
                         Temporal Noun Phrase
==========================================================================*/

tnp1([qclause:D,num:Num,sem:TNP1]) -->
                         det([type:D,sem:Det]),tnbar1([type:D,num:Num,sem:TnBar]),
                         {var_replace(TnBar,TnBar1),beta(TnBar1@Det,TNP1)}.
tnp0([qclause:D,num:Num,sem:TNP1]) -->
                         det([type:D,sem:Det]),tnbar0([type:D,num:Num,sem:TnBar]),
                         {var_replace(Det,Det1),beta(Det1@TnBar,TNP1)}.
tnp0([qclause:init_exists,num:Num,sem:NumS]) -->
                         num([sem:NumS]), tnbar0([type:_,num:Num,sem:_]).
tnbar1([type:_,num:Num,sem:TNBAR]) -->
                         tn([num:Num,sem:A]), tncoord([sem:B]), tn([num:Num,sem:C]),
                         {var_replace(B,B1),beta(B1@A,S),beta(S@C,TNBAR)}.
tnbar0([type:def,num:Num,sem:TNBAR]) -->
                         tAdj([sem:Tadj]), tn([num:Num,sem:Tn]),
                         {var_replace(Tadj,Tadj1),beta(Tadj1@Tn,TNBAR)}.
tnbar0([type:_,num:Num,sem:TNBAR]) -->
                          adj([sem:Adj]), tnbar0([type:_,num:Num,sem:Tn]),
                         {var_replace(Adj,Adj1),beta(Adj1@Tn,TNBAR)}.
tnbar0([type:_,num:Num,sem:Tn]) --> tn([num:Num,sem:Tn]).

/*==========================================================================
                         Non-Temporal Noun Phrase
==========================================================================*/
np1([num:Num,sem:NP1]) -->
        np0([num:Num,sem:A]),npcoord([sem:B]), np0([num:Num,sem:C]),
                          {var_replace(B,B1),beta(B1@A,S),beta(S@C,NP1)}.
```

```
np0([num:Num,sem:PN]) -->
                        det([type:def,sem:_]),  pn_def([num:Num,sem:PN]).
np0([num:Num,sem:PN])-->  pn([num:Num,sem:PN]).


/*=====================================================================
                        Predicate  Phrases
=====================================================================*/


ibar([num:Num,mclause:M,sem:NegP])--> aux([num:Num]),
                                      negP([symbol:[],mclause:M,sem:NegP]).
ibar([num:Num,mclause:M,sem:VP])  -->  aux([num:Num]),
                                       vp([symbol:[],mclause:M,sem:VP]).
ibar([num:Num,mclause:M,sem:NegP])--> main_verb([symbol:V,num:Num]),
                                      negP([symbol:V,mclause:M,sem:NegP]).
ibar([num:Num,mclause:M,sem:VP])-->   main_verb([symbol:V,num:Num]),
                                      vp([symbol:V,mclause:M,sem:VP]).
ibar([num:Num,mclause:M,sem:VP])-->   i([symbol:V,num:Num]),
                                      vp([symbol:V,mclause:M,sem:VP]).
negP([symbol:V,mclause:NewM,sem:NegP])--> neg([sem:Neg]),
                                          vp([symbol:V,mclause:M,sem:VP]),
                                          {quantifier_modification(M,NewM),
                                          var_replace(Neg,Neg1),beta(Neg1@VP,NegP)}.
vp([symbol:V,mclause:M,sem:VP])-->  extraAux, v([symbol:V,mclause:M,sem:VP]).
vp([symbol:V,mclause:M,sem:VP])-->  adv, v([symbol:V,mclause:M,sem:VP]).
vp([symbol:V,mclause:M,sem:VP])-->  v([symbol:V,mclause:M,sem:VP]),adv.
vp([symbol:V,mclause:M,sem:VP])-->  adv, extraAux, v([symbol:V,mclause:M,sem:VP]).
vp([symbol:V,mclause:M,sem:VP])-->  v([symbol:V,mclause:M,sem:VP]).
v([symbol:V,mclause:M,sem:IV])-->   iv([symbol:V,mclause:M,sem:IV]).
v([symbol:V,mclause:M,sem:Adj]) --> iv_adj([symbol:V,mclause:_,sem:_]),
                                     adj([type:M,sem:Adj]).
v([symbol:V,mclause:M,sem:VSem])-->  iv_adj([symbol:V,mclause:M,sem:IV]),
                                     adj([sem:Adj]),
                                     {var_replace(IV,IV1),beta(IV1@Adj,VSem)}.
v([symbol:V,mclause:M,sem:VSem])-->  tv([symbol:V,mclause:M,sem:TV]),
                                     np0([num:_,sem:NP]),
                                     {var_replace(TV,TV1),beta(TV1@NP,VSem)}.
v([symbol:V,mclause:M,sem:VSem])-->  tv([symbol:V,mclause:M,sem:TV]), [to],
                                     np0([num:_,sem:NP]),
                                     {var_replace(TV,TV1),beta(TV1@NP,VSem)}.
v([symbol:V,mclause:M,sem:DTV3])-->  dtv([symbol:V,mclause:M,sem:DTV]),
                                     np0([num:_,sem:NP1]),
                                     np0([num:_,sem:NP2]),
                   {var_replace(DTV,DTV1),beta(DTV1@NP1,DTV2),beta(DTV2@NP2,DTV3)}.
```

```
/*==========================================================================
                          Temporal preposition Phrase
==========================================================================*/

tpp([mclause:M,sem:TPP]) --->   tpn([qclause:Q,mclause:M,sem:TP]),
                                tnp1([qclause:Q,num:_,sem:NP]),
                                {var_replace(NP,NP1),beta(NP1@TP,TPP)}.
tpp([mclause:M,sem:TPP]) --->   tpn([qclause:Q,mclause:M,sem:TP]),
                                tnp0([qclause:Q,num:_,sem:NP]),
                                {var_replace(TP,TP1),beta(TP1@NP,TPP)}.
tpp([mclause:M,sem:TPP]) --->   tps([qclause:_,mclause:M,sem:TP]),
                                removeQ([mclause:_,sem:IP]),
                                {var_replace(TP,TP1),beta(TP1@IP,TPP)}.
removeQ([mclause:_,sem:E]) --->   ip0([mclause:_,sem:exists(E) during 'T']).
removeQ([mclause:_,sem:E]) --->   ip0([mclause:_,sem:forall(E)]).
tpps([sem:TPP])--->   tpp([mclause:_,sem:TPP]).
tpps([sem:TPP]) --->  tpp([mclause:_,sem:TPP1]), tpp([mclause:_,sem:TPP2]),
                      {var_replace(TPP2,TPP21),beta(TPP21@TPP1,TPP)}.
/*==========================================================================
                          Punctuataion
==========================================================================*/
punctuat ---> [then].
punctuat ---> [','].
punctuat ---> [','],[then].
punctuat ---> [ ].

/*==========================================================================
                          Lexical Rules for Closed class
==========================================================================*/
det([type:Type,sem:Det])--->
   {lexEntry(det,[syntax:Word,type:Type])}, Word,
   {semLex(det,[type:Type,sem:Det])}.

aux([num:Num])--->
   {lexEntry(aux,[syntax:Word,num:Num])}, Word.

main_verb([symbol:be,num:NUM]) --->
  {lexEntry(be_verb,[syntax:Word,num:NUM])}, [Word].

main_verb([symbol:have,num:NUM]) --->
  {lexEntry(possession_verb,[syntax:Word,num:NUM])}, [Word].

extraAux   --->   {lexEntry(extraAux,[syntax:Word])}, Word.
```

```
neg([sem:Neg])-->
    {lexEntry(neg,[syntax:Word])}, Word,
    {semLex(neg,[sem:Neg])}.

tAdj([sem:Sem])-->
    {lexEntry(tadj,[symbol:Sym,syntax:Word])}, Word,
    {semLex(tadj,[symbol:Sym,sem:Sem])}.

tncoord([sem:Sem])-->
    {lexEntry(coord,[syntax:Word,type:Type])}, Word,
    {semLex(tncoord,[type:Type,sem:Sem])}.

ipcoord([sem:Sem])-->
    {lexEntry(coord,[syntax:Word,type:Type])}, Word,
    {semLex(ipcoord,[type:Type,sem:Sem])}.

npcoord([sem:Sem])-->
    {lexEntry(coord,[syntax:Word,type:Type])}, Word,
    {semLex(npcoord,[type:Type,sem:Sem])}.

tpn([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [within].
tpn([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [within].
tpn([qclause:init_exists,mclause:exists,sem:lbd(n,lbd(p,n repeat p))])--> [within].
tpn([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [at].
tpn([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [at].
tpn([qclause:init_exists,mclause:exists,sem:lbd(n,lbd(p,n repeat p))])--> [at].
tpn([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])--> [at].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])--> [at].
tpn([qclause:init_exists,mclause:forall,sem:lbd(n,lbd(p,n repeat p))])--> [at].
tpn([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [at,least].
tpn([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [at,least].
tpn([qclause:init_exists,mclause:exists,sem:lbd(n,lbd(p,n repeat p))])--> [at,least].
tpn([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])--> [at,least].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])--> [at,least].
tpn([qclause:init_exists,mclause:forall,sem:lbd(n,lbd(p,n repeat p))])--> [at,least].
tpn([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [during].
tpn([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [during].
tpn([qclause:init_exists,mclause:exists,sem:lbd(n,lbd(p,n repeat p))])--> [during].
tpn([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])--> [during].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])--> [during].
tpn([qclause:init_exists,mclause:forall,sem:lbd(n,lbd(p,n repeat p))])--> [during].
tpn([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [on].
tpn([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])--> [on].
tpn([qclause:init_exists,mclause:exists,sem:lbd(n,lbd(p,n repeat p))])--> [on].
tpn([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])—> [on].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])—> [on].
tpn([qclause:init_exists,mclause:forall,sem:lbd(n,lbd(p,n repeat p))])—> [on].
tpn([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])—> [for].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])—> [for].
tpn([qclause:init_exists,mclause:forall,sem:lbd(n,lbd(p,n repeat p))])—> [for].
tpn([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])—> [in].
tpn([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,p1 during p2))])—> [in].
tpn([qclause:init_exists,mclause:exists,sem:lbd(n,lbd(p,n repeat p))])—> [in].
tpn([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])—> [throughout].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 during p2))])—> [throughout].
tpn([qclause:init_exists,mclause:forall,sem:lbd(n,lbd(p,n repeat p))])—> [throughout].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 before p2))])—> [until].
tpn([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,p1 before p2))])—> [by].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 before p2))])—> [since].
tpn([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,p1 before p2))])—> [before].
tpn([qclause:forall,mclause:init_exists,sem:lbd(p1,lbd(p2,p1 before p2))])—> [before].
tpn([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,p1 after p2))])—> [after].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 after p2))])—> [after].
tpn([qclause:forall,mclause:init_exists,sem:lbd(p1,lbd(p2,p1 after p2))])—> [after].
tpn([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,p1 until_after p2))])—> [until,after].
tps([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,forall(p1) during p2))])—> [when].
tps([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,forall(p1) during p2))])—> [when].
tps([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,forall(p1) during p2))])-->[whilst].
tps([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,forall(p1) during p2))])-->[whilst].
tps([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,forall(p1) during p2))])—> [while].
tps([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,forall(p1) during p2))])—> [while].
tps([qclause:forall,mclause:exists,sem:lbd(p1,lbd(p2,forall(p1) during p2))])—>
    [whenever].
tps([qclause:forall,mclause:forall,sem:lbd(p1,lbd(p2,forall(p1) during p2))])—>
    [whenever].
tps([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,def(p1) before p2))])—> [until].
tps([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,def(p1) before p2))])—> [before].
tps([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,def(p1) before p2))])—> [by,the,time].
tps([qclause:def,mclause:exists,sem:lbd(p1,lbd(p2,def(p1) after p2))])—> [after].
tps([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,def(p1) after p2))])—> [after].
tps([qclause:forall,mclause:init_exists,sem:lbd(p1,lbd(p2,forall(p1) after p2))])—> [
    once].
tps([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,def(p1) after p2))])—> [since].
tps([qclause:def,mclause:forall,sem:lbd(p1,lbd(p2,def(p1) until_after p2))])—> [until,
    after].

if([sem:lbd(p1,lbd(p2,p1 during p2))])  —>  [if].
either([sem:lbd(p1,lbd(p2, p1  v p2))])  —>  [either].
```

```
/*=====================================================================
                         Lexical Rules for Open Class
=====================================================================*/
pn([num:Num,sem:Sem])-->
   {lexEntry(pn,[symbol:Sym,num:Num,syntax:Word])}, Word,
   {semLex(pn,[symbol:Sym,sem:Sem])}.

pn([num:_,sem:Sem])-->
   {lexEntry(number,[symbol:Sym,syntax:Word])}, Word,
   {semLex(pn,[symbol:Sym,sem:Sem])}.

pn_def([num:Num,sem:Sem])-->
   {lexEntry(pn_def,[symbol:Sym,num:Num,syntax:Word])}, Word,
   {semLex(pn,[symbol:Sym,sem:Sem])}.

tn([num:Num,sem:Sem])-->
   {lexEntry(tnoun,[symbol:Sym,num:Num,syntax:Word])}, Word,
   {semLex(tnoun,[symbol:Sym,sem:Sem])}.

num([sem:N]) -->
   {lexEntry(number,[symbol:Sym,syntax:Word])}, Word,
   {semLex(number,[symbol:Sym,sem:N])}.

i([symbol:Root,num:sg]) -->
  [Word], {morph_type(Word, Root,-s), lexEntry(iv,[syntax:Root])}.
i([symbol:Root,num:_]) -->
  [Word], {morph_type(Word, Root,none), lexEntry(iv,[syntax:Root])}.
i([symbol:Root,num:_]) -->
  [Word], {morph_type(Word, Root,-ed), lexEntry(iv,[syntax:Root])}.

i([symbol:Root,num:sg]) -->
  [Word], {morph_type(Word, Root,-s), lexEntry(tv,[syntax:Root])}.
i([symbol:Root,num:_]) -->
  [Word], {morph_type(Word, Root,none), lexEntry(tv,[syntax:Root])}.
i([symbol:Root,num:_]) -->
  [Word], {morph_type(Word, Root,-ed), lexEntry(tv,[syntax:Root])}.

i([symbol:Root,num:sg]) -->
  [Word], {morph_type(Word, Root,-s), lexEntry(dtv,[syntax:Root])}.
i([symbol:Root,num:_]) -->
  [Word], {morph_type(Word, Root,none), lexEntry(dtv,[syntax:Root])}.
i([symbol:Root,num:_]) -->
  [Word], {morph_type(Word, Root,-ed), lexEntry(dtv,[syntax:Root])}.
```

```
iv([symbol:[],mclause:M,sem:Sem]) -->
    [Word],
    {morph_type(Word,Root,Suffix),
    grammatical_aspect(Suffix,GrAsp),
    aspectual_class(Root,Type),
    quantifier_type(GrAsp,Type,M),
    lexEntry(iv,[syntax:Root]),
    semLex(iv,[symbol:Root,type:M,sem:Sem])}.

iv([symbol:Word,mclause:M,sem:Sem]) -->
    [], {Word \= [], morph_type(Word,Root,Suffix),
    grammatical_aspect(Suffix,GrAsp),
    aspectual_class(Root,Type),
    quantifier_type(GrAsp,Type,M),
    lexEntry(iv,[syntax:Root]),
    semLex(iv,[symbol:Root,type:M,sem:Sem])}.

iv_adj([symbol:[],mclause:M,sem:Sem]) -->
    [Word],
    {morph_type(Word,Root,Suffix),
    grammatical_aspect(Suffix,GrAsp),
    aspectual_class(Root,Type),
    quantifier_type(GrAsp,Type,M),
    lexEntry(iv,[syntax:Root]),
    semLex(iv_adj,[symbol:Root,type:M,sem:Sem])}.

iv_adj([symbol:Word,mclause:M,sem:Sem]) -->
    [], {Word \= [], morph_type(Word,Root,Suffix),
    grammatical_aspect(Suffix,GrAsp),
    aspectual_class(Root,Type),
    quantifier_type(GrAsp,Type,M),
    lexEntry(iv,[syntax:Root]),
    semLex(iv_adj,[symbol:Root,type:M,sem:Sem])}.

tv([symbol:[],mclause:M,sem:Sem]) -->
    [Word],
    {morph_type(Word,Root,Suffix),
    grammatical_aspect(Suffix,GrAsp),
    aspectual_class(Root,Type),
    quantifier_type(GrAsp,Type,M),
    lexEntry(tv,[syntax:Root]),
    semLex(tv,[symbol:Root,type:M,sem:Sem])}.
```

```
tv([symbol:Word, mclause:M, sem:Sem]) -->
    [], {Word \= [], morph_type(Word, Root, Suffix),
    grammatical_aspect(Suffix, GrAsp),
    aspectual_class(Root, Type),
    quantifier_type(GrAsp, Type, M),
    lexEntry(tv, [syntax:Root]),
    semLex(tv, [symbol:Root, type:M, sem:Sem])}.

dtv([symbol:[], mclause:M, sem:Sem]) -->
    [Word],
    {morph_type(Word, Root, Suffix),
    grammatical_aspect(Suffix, GrAsp),
    aspectual_class(Root, Type),
    quantifier_type(GrAsp, Type, M),
    lexEntry(dtv, [syntax:Root]),
    semLex(dtv, [symbol:Root, type:M, sem:Sem])}.

dtv([symbol:Word, mclause:M, sem:Sem]) -->
    [], {Word \= [], morph_type(Word, Root, Suffix),
    grammatical_aspect(Suffix, GrAsp),
    aspectual_class(Root, Type),
    quantifier_type(GrAsp, Type, M),
    lexEntry(dtv, [syntax:Root]),
    semLex(dtv, [symbol:Root, type:M, sem:Sem])}.

adv -->
    {lexEntry(adv, [syntax:Word])}, Word.

adj([type:exists, sem:Sem])-->
    {lexEntry(adj_Stable, [syntax:Word])}, Word,
    {semLex(adj_Stable, [sem:Sem])}.

adj([type:init_exists, sem:Sem])-->
    {lexEntry(adj_Stable, [syntax:Word])}, Word,
    {semLex(adj_Stable, [sem:Sem])}.

adj([sem:Sem])-->
    {lexEntry(adj, [symbol:Sym, syntax:Word])}, Word,
    {semLex(adj, [symbol:Sym, sem:Sem])}.
```

```
/*=========================================================================
                    Lexical Entry for Closed Class                       */
/*=========================================================================
                    Determiners
=========================================================================*/
lexEntry(det,[syntax:[any],type:forall]).
lexEntry(det,[syntax:[every],type:forall]).
lexEntry(det,[syntax:[all],type:forall]).
lexEntry(det,[syntax:[a],type:exists]).
lexEntry(det,[syntax:[an],type:exists]).
lexEntry(det,[syntax:[some],type:exists]).
lexEntry(det,[syntax:[the],type:def]).
/*=========================================================================
                    Auxiliary Verbs
=========================================================================*/
lexEntry(aux,[syntax:[is],num:sg]).
lexEntry(aux,[syntax:[are],num:pl]).
lexEntry(aux,[syntax:[was],num:sg]).
lexEntry(aux,[syntax:[were],num:pl]).
lexEntry(aux,[syntax:[has],num:sg]).
lexEntry(aux,[syntax:[have],num:sg]).
lexEntry(aux,[syntax:[had],num:pl]).
lexEntry(aux,[syntax:[must],num:_]).
lexEntry(aux,[syntax:[may],num:_]).
lexEntry(aux,[syntax:[might],num:_]).
lexEntry(aux,[syntax:[can],num:_]).
lexEntry(aux,[syntax:[could],num:_]).
lexEntry(aux,[syntax:[will],num:_]).
lexEntry(aux,[syntax:[would],num:_]).
lexEntry(aux,[syntax:[shall],num:_]).
lexEntry(aux,[syntax:[should],num:_]).
/*=========================================================================
                    Multiple auxiliaries
=========================================================================*/
lexEntry(extraAux,[syntax:[be]]).
lexEntry(extraAux,[syntax:[been]]).
lexEntry(extraAux,[syntax:[being]]).
lexEntry(extraAux,[syntax:[have,been]]).
lexEntry(extraAux,[syntax:[have,being]]).
lexEntry(extraAux,[syntax:[been,being]]).
/*=========================================================================
                    Negation
=========================================================================*/
lexEntry(neg,[syntax:[no]]).
lexEntry(neg,[syntax:[not]]).
lexEntry(neg,[syntax:[never]]).
/*=====================================================================
                    Temporal Adjectives
=====================================================================*/
lexEntry(tadj,[symbol:f,syntax:[first]]).
lexEntry(tadj,[symbol:l,syntax:[last]]).
/*=====================================================================
                    Coordinations
=====================================================================*/
lexEntry(coord,[syntax:[and],type:conj]).
lexEntry(coord,[syntax:[or],type:disj]).
/*=====================================================================
                    BE Verbs
=====================================================================*/
lexEntry(be_verb,[syntax:is,num:sg]).
lexEntry(be_verb,[syntax:are,num:pl]).
lexEntry(be_verb,[syntax:was,num:sg]).
lexEntry(be_verb,[syntax:were,num:pl]).
/*=====================================================================
                    Possession Verbs
=====================================================================*/
lexEntry(possession_verb,[syntax:has,num:sg]).
lexEntry(possession_verb,[syntax:have,num:pl]).
lexEntry(possession_verb,[syntax:had,num:_]).

/*=====================================================================
                    Lexical Entry for Open Class                    */
/*=====================================================================
                    Proper Names
=====================================================================*/
lexEntry(pn,[symbol:'X',num:_,syntax:['X']]).
lexEntry(pn,[symbol:'Z',num:_,syntax:['Z']]).
lexEntry(pn,[symbol:'Awid',num:_,syntax:['Awid']]).
lexEntry(pn,[symbol:'Awvalid',num:_,syntax:['Awvalid']]).
lexEntry(pn,[symbol:'Awready',num:_,syntax:['Awready']]).
lexEntry(pn,[symbol:'Awprot',num:_,syntax:['Awprot']]).
lexEntry(pn,[symbol:'Tvalid',num:_,syntax:['Tvalid']]).
lexEntry(pn,[symbol:'WRAP',num:_,syntax:['WRAP']]).
lexEntry(pn,[symbol:'Awaddr',num:_,syntax:['Awaddr']]).
lexEntry(pn,[symbol:'Wready',num:_,syntax:['Wready']]).
lexEntry(pn,[symbol:'Arcache',num:_,syntax:['Arcache']]).
lexEntry(pn,[symbol:'Wid',num:_,syntax:['Wid']]).
lexEntry(pn,[symbol:'Arcache[1]',num:_,syntax:['Arcache[1]']]).
lexEntry(pn,[symbol:'Arcache[3:2]',num:_,syntax:['Arcache[3:2]']]).
lexEntry(pn,[symbol:'Arprot',num:_,syntax:['Arprot']]).
lexEntry(pn,[symbol:'Arsize',num:_,syntax:['Arsize']]).
lexEntry(pn,[symbol:'Arburst',num:_,syntax:['Arburst']]).
lexEntry(pn,[symbol:'RID',num:_,syntax:['RID']]).
lexEntry(pn,[symbol:'Rready',num:_,syntax:['Rready']]).
lexEntry(pn,[symbol:'Rvalid',num:_,syntax:['Rvalid']]).
lexEntry(pn,[symbol:'CsysReq',num:_,syntax:['CsysReq']]).
lexEntry(pn,[symbol:'CsysAck',num:_,syntax:['CsysAck']]).
lexEntry(pn,[symbol:'Cactive',num:_,syntax:['Cactive']]).
lexEntry(pn,[symbol:'Arlock',num:_,syntax:['Arlock']]).
lexEntry(pn,[symbol:'Rdata',num:_,syntax:['Rdata']]).
lexEntry(pn,[symbol:'Bresp',num:_,syntax:['Bresp']]).
lexEntry(pn,[symbol:'Wstrb',num:_,syntax:['Wstrb']]).
lexEntry(pn,[symbol:'Wdata',num:_,syntax:['Wdata']]).
lexEntry(pn,[symbol:'Araddr',num:_,syntax:['Araddr']]).
lexEntry(pn,[symbol:'Arlen',num:_,syntax:['Arlen']]).
lexEntry(pn,[symbol:'Arvalid',num:_,syntax:['Arvalid']]).
lexEntry(pn,[symbol:'Wlast',num:_,syntax:['Wlast']]).
lexEntry(pn,[symbol:'Arready',num:_,syntax:['Arready']]).
lexEntry(pn,[symbol:'Arid',num:_,syntax:['Arid']]).
lexEntry(pn,[symbol:'Awlen',num:_,syntax:['Awlen']]).
lexEntry(pn,[symbol:'Awdomain',num:_,syntax:['Awdomain']]).
lexEntry(pn,[symbol:'Ardomain',num:_,syntax:['Ardomain']]).
lexEntry(pn,[symbol:'Awuser',num:_,syntax:['Awuser']]).
lexEntry(pn,[symbol:'Awuser_Width',num:_,syntax:['Awuser_Width']]).
lexEntry(pn,[symbol:'Awsize',num:_,syntax:['Awsize']]).
lexEntry(pn,[symbol:'Bvalid',num:_,syntax:['Bvalid']]).
lexEntry(pn,[symbol:'Bready',num:_,syntax:['Bready']]).
lexEntry(pn,[symbol:'Wvalid',num:_,syntax:['Wvalid']]).
lexEntry(pn,[symbol:'Awburst',num:_,syntax:['Awburst']]).
lexEntry(pn,[symbol:'Awlock',num:_,syntax:['Awlock']]).
lexEntry(pn,[symbol:'Awcache',num:_,syntax:['Awcache']]).
lexEntry(pn,[symbol:'Awcache[1]',num:_,syntax:['Awcache[1]']]).
lexEntry(pn,[symbol:'Awcache[3:2]',num:_,syntax:['Awcache[3:2]']]).
lexEntry(pn,[symbol:'Aresetn',num:_,syntax:['Aresetn']]).
lexEntry(pn,[symbol:'BID',num:_,syntax:['BID']]).
lexEntry(pn,[symbol:'Arestn',num:_,syntax:['Arestn']]).
lexEntry(pn,[symbol:'Data_Width_Bytes',num:_,syntax:['Data_Width_Bytes']]).
lexEntry(pn,[symbol:'Rresp',num:_,syntax:['Rresp']]).
lexEntry(pn,[symbol:'Rlast',num:_,syntax:['Rlast']]).
lexEntry(pn,[symbol:'Addr_Width',num:_,syntax:['Addr_Width']]).
lexEntry(pn,[symbol:'Wdepth',num:_,syntax:['Wdepth']]).
lexEntry(pn,[symbol:'MaxrBursts',num:_,syntax:['MaxrBursts']]).
lexEntry(pn,[symbol:'MaxwBursts',num:_,syntax:['MaxwBursts']]).
lexEntry(pn,[symbol:'Awsnoop',num:_,syntax:['Awsnoop']]).
lexEntry(pn,[symbol:'Awbar',num:_,syntax:['Awbar']]).
lexEntry(pn,[symbol:'Awqos',num:_,syntax:['Awqos']]).
lexEntry(pn,[symbol:'Awregion',num:_,syntax:['Awregion']]).
lexEntry(pn,[symbol:'S_OK',num:_,syntax:['S_OK']]).
lexEntry(pn,[symbol:'Buser_Width',num:_,syntax:['Buser_Width']]).
lexEntry(pn,[symbol:'Buser',num:_,syntax:['Buser']]).
lexEntry(pn,[symbol:'Aruser_Width',num:_,syntax:['Aruser_Width']]).
lexEntry(pn,[symbol:'Aruser',num:_,syntax:['Aruser']]).
lexEntry(pn,[symbol:'MRMD',num:_,syntax:['MRMD']]).
lexEntry(pn,[symbol:'Arqos',num:_,syntax:['Arqos']]).
lexEntry(pn,[symbol:'Arregion',num:_,syntax:['Arregion']]).
lexEntry(pn,[symbol:'Ruser_Width',num:_,syntax:['Ruser_Width']]).
lexEntry(pn,[symbol:'Ruser',num:_,syntax:['Ruser']]).
lexEntry(pn,[symbol:'Data_width',num:_,syntax:['Data_width']]).
lexEntry(pn,[symbol:'Exmon_Width',num:_,syntax:['Exmon_Width']]).
lexEntry(pn,[symbol:'TID',num:_,syntax:['TID']]).
lexEntry(pn,[symbol:'Tready',num:_,syntax:['Tready']]).
lexEntry(pn,[symbol:'Tdest',num:_,syntax:['Tdest']]).
lexEntry(pn,[symbol:'Tdata',num:_,syntax:['Tdata']]).
lexEntry(pn,[symbol:'Tstrb',num:_,syntax:['Tstrb']]).
lexEntry(pn,[symbol:'Tlast',num:_,syntax:['Tlast']]).
lexEntry(pn,[symbol:'Tkeep',num:_,syntax:['Tkeep']]).
lexEntry(pn,[symbol:'Tuser',num:_,syntax:['Tuser']]).
lexEntry(pn,[symbol:'Dest_Width',num:_,syntax:['Dest_Width']]).
lexEntry(pn,[symbol:'User_Width',num:_,syntax:['User_Width']]).
lexEntry(pn,[symbol:'Arsnoop',num:_,syntax:['Arsnoop']]).
lexEntry(pn,[symbol:'Arbar',num:_,syntax:['Arbar']]).
lexEntry(pn,[symbol:'Resp[3]',num:_,syntax:['Resp[3]']]).
lexEntry(pn,[symbol:'Resp[2]',num:_,syntax:['Resp[2]']]).
lexEntry(pn,[symbol:'Rack',num:_,syntax:['Rack']]).
lexEntry(pn,[symbol:'Wack',num:_,syntax:['Wack']]).
lexEntry(pn,[symbol:'AW',num:_,syntax:['AW']]).
lexEntry(pn,[symbol:'Acvalid',num:_,syntax:['Acvalid']]).
lexEntry(pn,[symbol:'Crvalid',num:_,syntax:['Crvalid']]).
lexEntry(pn,[symbol:'Cdvalid',num:_,syntax:['Cdvalid']]).
lexEntry(pn,[symbol:'MaxcBursts',num:_,syntax:['MaxcBursts']]).
lexEntry(pn,[symbol:'Max_Barriers',num:_,syntax:['Max_Barriers']]).
lexEntry(pn,[symbol:'Acready',num:_,syntax:['Acready']]).
lexEntry(pn,[symbol:'Wuser',num:_,syntax:['Wuser']]).
lexEntry(pn,[symbol:'Wuser_Width',num:_,syntax:['Wuser_Width']]).
lexEntry(pn,[symbol:'ID_Width',num:_,syntax:['ID_Width']]).
lexEntry(pn,[symbol:'Acsnoop',num:_,syntax:['Acsnoop']]).
lexEntry(pn,[symbol:'Acprot',num:_,syntax:['Acprot']]).
lexEntry(pn,[symbol:'Crresp[2]',num:_,syntax:['Crresp[2]']]).
lexEntry(pn,[symbol:'Crresp',num:_,syntax:['Crresp']]).
lexEntry(pn,[symbol:'Crresp[4:0]',num:_,syntax:['Crresp[4:0]']]).
lexEntry(pn,[symbol:'Crready',num:_,syntax:['Crready']]).
lexEntry(pn,[symbol:'Cdlast',num:_,syntax:['Cdlast']]).
lexEntry(pn,[symbol:'Cddata',num:_,syntax:['Cddata']]).
lexEntry(pn,[symbol:'Cdready',num:_,syntax:['Cdready']]).
lexEntry(pn,[symbol:'Acaddr',num:_,syntax:['Acaddr']]).
lexEntry(pn,[symbol:'CD_Data_Width',num:_,syntax:['CD_Data_Width']]).
lexEntry(pn,[symbol:'Addr',num:_,syntax:['Addr']]).
lexEntry(pn,[symbol:'Mcmd',num:_,syntax:['Mcmd']]).
lexEntry(pn,[symbol:'AtomicLength',num:_,syntax:['AtomicLength']]).
lexEntry(pn,[symbol:'Atomiclength_Wdth',num:_,syntax:['Atomiclength_Wdth']]).
lexEntry(pn,[symbol:'BCST',num:_,syntax:['BCST']]).
lexEntry(pn,[symbol:'MBlockHeight',num:_,syntax:['MBlockHeight']]).
lexEntry(pn,[symbol:'BlockHeight_Wdth',num:_,syntax:['BlockHeight_Wdth']]).
lexEntry(pn,[symbol:'BlockStride',num:_,syntax:['BlockStride']]).
lexEntry(pn,[symbol:'Broadcast_Enable',num:_,syntax:['Broadcast_Enable']]).
lexEntry(pn,[symbol:'Burst_Aligned',num:_,syntax:['Burst_Aligned']]).
lexEntry(pn,[symbol:'BurstLength',num:_,syntax:['BurstLength']]).
lexEntry(pn,[symbol:'MBlockStride[0]',num:_,syntax:['MBlockStride[0]']]).
lexEntry(pn,[symbol:'MBlockStride[1:0]',num:_,syntax:['MBlockStride[1:0]']]).
lexEntry(pn,[symbol:'MBlockStride[2:0]',num:_,syntax:['MBlockStride[2:0]']]).
lexEntry(pn,[symbol:'MBlockStride[3:0]',num:_,syntax:['MBlockStride[3:0]']]).
lexEntry(pn,[symbol:'Maddr[0]',num:_,syntax:['Maddr[0]']]).
lexEntry(pn,[symbol:'Maddr[1:0]',num:_,syntax:['Maddr[1:0]']]).
lexEntry(pn,[symbol:'Maddr[2:0]',num:_,syntax:['Maddr[2:0]']]).
lexEntry(pn,[symbol:'Maddr[3:0]',num:_,syntax:['Maddr[3:0]']]).
lexEntry(pn,[symbol:'BurstLength_Wdth',num:_,syntax:['BurstLength_Wdth']]).
lexEntry(pn,[symbol:'BurstPrecise',num:_,syntax:['BurstPrecise']]).
lexEntry(pn,[symbol:'BurstSeq',num:_,syntax:['BurstSeq']]).
lexEntry(pn,[symbol:'BurstSeq_Blck_Enable',num:_,syntax:['BurstSeq_Blck_Enable']]).
lexEntry(pn,[symbol:'BurstSeq_Dflt1_Enable',num:_,syntax:['BurstSeq_Dflt1_Enable']]).
lexEntry(pn,[symbol:'BurstSeq_Dflt2_Enable',num:_,syntax:['BurstSeq_Dflt2_Enable']]).
lexEntry(pn,[symbol:'BurstSeq_Enable',num:_,syntax:['BurstSeq_Enable']]).
lexEntry(pn,[symbol:'BurstSeq_Incr_Enable',num:_,syntax:['BurstSeq_Incr_Enable']]).
lexEntry(pn,[symbol:'BurstSeq_Strm_Enable',num:_,syntax:['BurstSeq_Strm_Enable']]).
lexEntry(pn,[symbol:'BurstSeq_Unkn_Enable',num:_,syntax:['BurstSeq_Unkn_Enable']]).
lexEntry(pn,[symbol:'BurstSeq_Wrap_Enable',num:_,syntax:['BurstSeq_Wrap_Enable']]).
lexEntry(pn,[symbol:'BurstSeq_XOR_Enable',num:_,syntax:['BurstSeq_XOR_Enable']]).
lexEntry(pn,[symbol:'BurstSingleReq',num:_,syntax:['BurstSingleReq']]).
lexEntry(pn,[symbol:'ByteEn',num:_,syntax:['ByteEn']]).
lexEntry(pn,[symbol:'CmdAccept',num:_,syntax:['CmdAccept']]).
lexEntry(pn,[symbol:'ConnectCap',num:_,syntax:['ConnectCap']]).
lexEntry(pn,[symbol:'Connection',num:_,syntax:['Connection']]).
lexEntry(pn,[symbol:'Control',num:_,syntax:['Control']]).
lexEntry(pn,[symbol:'ControlBusy',num:_,syntax:['ControlBusy']]).
lexEntry(pn,[symbol:'ControlWr',num:_,syntax:['ControlWr']]).
lexEntry(pn,[symbol:'Corresponding_Exact',num:_,syntax:['Corresponding_Exact']]).
lexEntry(pn,[symbol:'DataAccept',num:_,syntax:['DataAccept']]).
lexEntry(pn,[symbol:'Datahandshake',num:_,syntax:['Datahandshake']]).
lexEntry(pn,[symbol:'DataLast',num:_,syntax:['DataLast']]).
lexEntry(pn,[symbol:'DataRowLast',num:_,syntax:['DataRowLast']]).
lexEntry(pn,[symbol:'Data_Wdth',num:_,syntax:['Data_Wdth']]).
lexEntry(pn,[symbol:'DLFT1',num:_,syntax:['DLFT1']]).
lexEntry(pn,[symbol:'DLFT2',num:_,syntax:['DLFT2']]).
lexEntry(pn,[symbol:'DFLT2',num:_,syntax:['DFLT2']]).
lexEntry(pn,[symbol:'SThreadBusy_Exact',num:_,syntax:['SThreadBusy_Exact']]).
lexEntry(pn,[symbol:'SThreadBusy_ExactCan',num:_,syntax:['SThreadBusy_ExactCan']]).
lexEntry(pn,[symbol:'MThreadBusy_Exact',num:_,syntax:['MThreadBusy_Exact']]).
lexEntry(pn,[symbol:'SDataThreadBusy_Exact',num:_,syntax:['SDataThreadBusy_Exact']]).
lexEntry(pn,[symbol:'Force_Aligned',num:_,syntax:['Force_Aligned']]).
lexEntry(pn,[symbol:'IDLE',num:_,syntax:['IDLE']]).
lexEntry(pn,[symbol:'INCR',num:_,syntax:['INCR']]).
lexEntry(pn,[symbol:'Jtag_Enable',num:_,syntax:['Jtag_Enable']]).
lexEntry(pn,[symbol:'Jtagtrst_Enable',num:_,syntax:['Jtagtrst_Enable']]).
lexEntry(pn,[symbol:'MAddr',num:_,syntax:['MAddr']]).
lexEntry(pn,[symbol:'MAddrSpace',num:_,syntax:['MAddrSpace']]).
lexEntry(pn,[symbol:'MAtomicLength',num:_,syntax:['MAtomicLength']]).
lexEntry(pn,[symbol:'MBlockHeight',num:_,syntax:['MBlockHeight']]).
lexEntry(pn,[symbol:'MBurstPrecise',num:_,syntax:['MBurstPrecise']]).
lexEntry(pn,[symbol:'MBlockStride',num:_,syntax:['MBlockStride']]).
lexEntry(pn,[symbol:'MBurstLength',num:_,syntax:['MBurstLength']]).
lexEntry(pn,[symbol:'MBurstSeq',num:_,syntax:['MBurstSeq']]).
lexEntry(pn,[symbol:'MBurstSingleReq',num:_,syntax:['MBurstSingleReq']]).
lexEntry(pn,[symbol:'MByteEn',num:_,syntax:['MByteEn']]).
lexEntry(pn,[symbol:'M_CON',num:_,syntax:['M_CON']]).
lexEntry(pn,[symbol:'MConnect',num:_,syntax:['MConnect']]).
lexEntry(pn,[symbol:'MConnID',num:_,syntax:['MConnID']]).
lexEntry(pn,[symbol:'MData',num:_,syntax:['MData']]).
lexEntry(pn,[symbol:'MDataByteEn',num:_,syntax:['MDataByteEn']]).
lexEntry(pn,[symbol:'MDataInfo',num:_,syntax:['MDataInfo']]).
lexEntry(pn,[symbol:'MDataInfoByte_Wdth',num:_,syntax:['MDataInfoByte_Wdth']]).
lexEntry(pn,[symbol:'MDataLast',num:_,syntax:['MDataLast']]).
lexEntry(pn,[symbol:'MDataRowLast',num:_,syntax:['MDataRowLast']]).
lexEntry(pn,[symbol:'MDataTagID',num:_,syntax:['MDataTagID']]).
lexEntry(pn,[symbol:'MDataThreadID',num:_,syntax:['MDataThreadID']]).
lexEntry(pn,[symbol:'MDataValid',num:_,syntax:['MDataValid']]).
lexEntry(pn,[symbol:'M_DISC',num:_,syntax:['M_DISC']]).
lexEntry(pn,[symbol:'M_OFF',num:_,syntax:['M_OFF']]).
lexEntry(pn,[symbol:'MError',num:_,syntax:['MError']]).
lexEntry(pn,[symbol:'MReqInfo',num:_,syntax:['MReqInfo']]).
lexEntry(pn,[symbol:'MReqLast',num:_,syntax:['MReqLast']]).
lexEntry(pn,[symbol:'MReqRowLast',num:_,syntax:['MReqRowLast']]).
lexEntry(pn,[symbol:'MReset',num:_,syntax:['MReset']]).
lexEntry(pn,[symbol:'MReset_n',num:_,syntax:['MReset_n']]).
lexEntry(pn,[symbol:'Reset',num:_,syntax:['Reset']]).
lexEntry(pn,[symbol:'MRespAccept',num:_,syntax:['MRespAccept']]).
lexEntry(pn,[symbol:'MTagID',num:_,syntax:['MTagID']]).
lexEntry(pn,[symbol:'MTagInOrder',num:_,syntax:['MTagInOrder']]).
lexEntry(pn,[symbol:'MThreadBusy',num:_,syntax:['MThreadBusy']]).
lexEntry(pn,[symbol:'MThreadID',num:_,syntax:['MThreadID']]).
lexEntry(pn,[symbol:'SThreadBusy_Pipelined',num:_,syntax:['SThreadBusy_Pipelined']]).
lexEntry(pn,[symbol:'MThreadBusy_Pipelined',num:_,syntax:['MThreadBusy_Pipelined']]).
lexEntry(pn,[symbol:'SDataThreadBusy_Pipelined',num:_,syntax:['SDataThreadBusy_Pipelined'
    ]]).
lexEntry(pn,[symbol:'RD',num:_,syntax:['RD']]).
lexEntry(pn,[symbol:'RDEX',num:_,syntax:['RDEX']]).
lexEntry(pn,[symbol:'RDL',num:_,syntax:['RDL']]).
lexEntry(pn,[symbol:'RDLwrc_Enable',num:_,syntax:['RDLwrc_Enable']]).
lexEntry(pn,[symbol:'Read_Enable',num:_,syntax:['Read_Enable']]).
lexEntry(pn,[symbol:'ReadEx',num:_,syntax:['ReadEx']]).
lexEntry(pn,[symbol:'ReadEx_Enable',num:_,syntax:['ReadEx_Enable']]).
lexEntry(pn,[symbol:'ReqData_Together',num:_,syntax:['ReqData_Together']]).
lexEntry(pn,[symbol:'ReqLast',num:_,syntax:['ReqLast']]).
lexEntry(pn,[symbol:'ReqRowLast',num:_,syntax:['ReqRowLast']]).
lexEntry(pn,[symbol:'RESP',num:_,syntax:['RESP']]).
lexEntry(pn,[symbol:'RespAccept',num:_,syntax:['RespAccept']]).
lexEntry(pn,[symbol:'RespInfo',num:_,syntax:['RespInfo']]).
lexEntry(pn,[symbol:'Res_ast',num:_,syntax:['Res_ast']]).
lexEntry(pn,[symbol:'RespRowLast',num:_,syntax:['RespRowLast']]).
lexEntry(pn,[symbol:'RiteNonPost_Enable',num:_,syntax:['RiteNonPost_Enable']]).
lexEntry(pn,[symbol:'SCmdAccept',num:_,syntax:['SCmdAccept']]).
lexEntry(pn,[symbol:'S_CON',num:_,syntax:['S_CON']]).
lexEntry(pn,[symbol:'SConnect',num:_,syntax:['SConnect']]).
lexEntry(pn,[symbol:'SData',num:_,syntax:['SData']]).
lexEntry(pn,[symbol:'SDataAccept',num:_,syntax:['SDataAccept']]).
lexEntry(pn,[symbol:'SDataInfo',num:_,syntax:['SDataInfo']]).
lexEntry(pn,[symbol:'SDataInfoByte_Wdth',num:_,syntax:['SDataInfoByte_Wdth']]).
lexEntry(pn,[symbol:'SDataInfo_Wdth',num:_,syntax:['SDataInfo_Wdth']]).
lexEntry(pn,[symbol:'SDataThreadBusy',num:_,syntax:['SDataThreadBusy']]).
lexEntry(pn,[symbol:'SError',num:_,syntax:['SError']]).
lexEntry(pn,[symbol:'SInterrupt',num:_,syntax:['SInterrupt']]).
lexEntry(pn,[symbol:'SReset',num:_,syntax:['SReset']]).
lexEntry(pn,[symbol:'SReset_n',num:_,syntax:['SReset_n']]).
lexEntry(pn,[symbol:'SResp',num:_,syntax:['SResp']]).
lexEntry(pn,[symbol:'SRespInfo',num:_,syntax:['SRespInfo']]).
lexEntry(pn,[symbol:'SRes_ast',num:_,syntax:['SRes_ast']]).
lexEntry(pn,[symbol:'SRespRowLast',num:_,syntax:['SRespRowLast']]).
lexEntry(pn,[symbol:'SRMD',num:_,syntax:['SRMD']]).
lexEntry(pn,[symbol:'STagID',num:_,syntax:['STagID']]).
lexEntry(pn,[symbol:'STagInOrder',num:_,syntax:['STagInOrder']]).
lexEntry(pn,[symbol:'Status',num:_,syntax:['Status']]).
lexEntry(pn,[symbol:'StatusBusy',num:_,syntax:['StatusBusy']]).
lexEntry(pn,[symbol:'StatusRd',num:_,syntax:['StatusRd']]).
lexEntry(pn,[symbol:'SThreadBusy',num:_,syntax:['SThreadBusy']]).
lexEntry(pn,[symbol:'SThreadID',num:_,syntax:['SThreadID']]).
lexEntry(pn,[symbol:'STRM',num:_,syntax:['STRM']]).
lexEntry(pn,[symbol:'SWait',num:_,syntax:['SWait']]).
lexEntry(pn,[symbol:'TagInOrder',num:_,syntax:['TagInOrder']]).
lexEntry(pn,[symbol:'UNKN',num:_,syntax:['UNKN']]).
lexEntry(pn,[symbol:'MDataInfo_Wdth',num:_,syntax:['MDataInfo_Wdth']]).
lexEntry(pn,[symbol:'MDataInfoByte_Wdth',num:_,syntax:['MDataInfoByte_Wdth']]).
lexEntry(pn,[symbol:'WR',num:_,syntax:['WR']]).
lexEntry(pn,[symbol:'WRC',num:_,syntax:['WRC']]).
lexEntry(pn,[symbol:'Write_Enable',num:_,syntax:['Write_Enable']]).
lexEntry(pn,[symbol:'WriteNonPost_Enable',num:_,syntax:['WriteNonPost_Enable']]).
lexEntry(pn,[symbol:'WRNP',num:_,syntax:['WRNP']]).
lexEntry(pn,[symbol:'MRespRowLast',num:_,syntax:['MRespRowLast']]).
lexEntry(pn,[symbol:'XOR',num:_,syntax:['XOR']]).
lexEntry(pn,[symbol:'Barrier',num:_,syntax:['Barrier']]).
lexEntry(pn,[symbol:'Data_Width',num:_,syntax:['Data','Width']]).
lexEntry(pn,[symbol:'AC_payload',num:_,syntax:['AC',payload]]).
lexEntry(pn,[symbol:'CD_payload',num:_,syntax:['CD',payload]]).
lexEntry(pn,[symbol:'CR_payload',num:_,syntax:['CR',payload]]).
lexEntry(pn,[symbol:'TUSER_payload',num:_,syntax:['TUSER',payload]]).
lexEntry(pn,[symbol:'RData_valid_byte_lanes',num:_,syntax:['RData',valid,byte,lanes]]).
lexEntry(pn,[symbol:'WData_valid_byte_lanes',num:_,syntax:['WData',valid,byte,lanes]]).

/*=============================================================================
        Proper Names that require a preceeding definite article
=============================================================================*/

lexEntry(pn_def,[symbol:'Awid',num:_,syntax:['Awid']]).
lexEntry(pn_def,[symbol:'Awvalid',num:_,syntax:['Awvalid']]).
lexEntry(pn_def,[symbol:'Awready',num:_,syntax:['Awready']]).
lexEntry(pn_def,[symbol:'Awprot',num:_,syntax:['Awprot']]).
lexEntry(pn_def,[symbol:'Tvalid',num:_,syntax:['Tvalid']]).
lexEntry(pn_def,[symbol:'WRAP',num:_,syntax:['WRAP']]).
lexEntry(pn_def,[symbol:'Awaddr',num:_,syntax:['Awaddr']]).
lexEntry(pn_def,[symbol:'Wready',num:_,syntax:['Wready']]).
lexEntry(pn_def,[symbol:'Arcache',num:_,syntax:['Arcache']]).
lexEntry(pn_def,[symbol:'Wid',num:_,syntax:['Wid']]).
lexEntry(pn_def,[symbol:'Arcache[1]',num:_,syntax:['Arcache[1]']]).
lexEntry(pn_def,[symbol:'Arcache[3:2]',num:_,syntax:['Arcache[3:2]']]).
lexEntry(pn_def,[symbol:'Arprot',num:_,syntax:['Arprot']]).
lexEntry(pn_def,[symbol:'Arsize',num:_,syntax:['Arsize']]).
lexEntry(pn_def,[symbol:'Arburst',num:_,syntax:['Arburst']]).
lexEntry(pn_def,[symbol:'RID',num:_,syntax:['RID']]).
lexEntry(pn_def,[symbol:'Rready',num:_,syntax:['Rready']]).
lexEntry(pn_def,[symbol:'Rvalid',num:_,syntax:['Rvalid']]).
lexEntry(pn_def,[symbol:'CsysReq',num:_,syntax:['CsysReq']]).
lexEntry(pn_def,[symbol:'CsysAck',num:_,syntax:['CsysAck']]).
lexEntry(pn_def,[symbol:'Cactive',num:_,syntax:['Cactive']]).
lexEntry(pn_def,[symbol:'Arlock',num:_,syntax:['Arlock']]).
lexEntry(pn_def,[symbol:'Rdata',num:_,syntax:['Rdata']]).
lexEntry(pn_def,[symbol:'Bresp',num:_,syntax:['Bresp']]).
lexEntry(pn_def,[symbol:'Wstrb',num:_,syntax:['Wstrb']]).
lexEntry(pn_def,[symbol:'Wdata',num:_,syntax:['Wdata']]).
lexEntry(pn_def,[symbol:'Araddr',num:_,syntax:['Araddr']]).
lexEntry(pn_def,[symbol:'Arlen',num:_,syntax:['Arlen']]).
lexEntry(pn_def,[symbol:'Arvalid',num:_,syntax:['Arvalid']]).
lexEntry(pn_def,[symbol:'Wlast',num:_,syntax:['Wlast']]).
lexEntry(pn_def,[symbol:'Arready',num:_,syntax:['Arready']]).
lexEntry(pn_def,[symbol:'Arid',num:_,syntax:['Arid']]).
lexEntry(pn_def,[symbol:'Awlen',num:_,syntax:['Awlen']]).
lexEntry(pn_def,[symbol:'Awdomain',num:_,syntax:['Awdomain']]).
lexEntry(pn_def,[symbol:'Ardomain',num:_,syntax:['Ardomain']]).
lexEntry(pn_def,[symbol:'Awuser',num:_,syntax:['Awuser']]).
lexEntry(pn_def,[symbol:'Awuser_Width',num:_,syntax:['Awuser_Width']]).
lexEntry(pn_def,[symbol:'Awsize',num:_,syntax:['Awsize']]).
lexEntry(pn_def,[symbol:'Bvalid',num:_,syntax:['Bvalid']]).
lexEntry(pn_def,[symbol:'Bready',num:_,syntax:['Bready']]).
lexEntry(pn_def,[symbol:'Wvalid',num:_,syntax:['Wvalid']]).
lexEntry(pn_def,[symbol:'Awburst',num:_,syntax:['Awburst']]).
lexEntry(pn_def,[symbol:'Awlock',num:_,syntax:['Awlock']]).
lexEntry(pn_def,[symbol:'Awcache',num:_,syntax:['Awcache']]).
lexEntry(pn_def,[symbol:'Awcache[1]',num:_,syntax:['Awcache[1]']]).
lexEntry(pn_def,[symbol:'Awcache[3:2]',num:_,syntax:['Awcache[3:2]']]).
lexEntry(pn_def,[symbol:'Aresetn',num:_,syntax:['Aresetn']]).
lexEntry(pn_def,[symbol:'BID',num:_,syntax:['BID']]).
lexEntry(pn_def,[symbol:'Arestn',num:_,syntax:['Arestn']]).
lexEntry(pn_def,[symbol:'Data_Width_Bytes',num:_,syntax:['Data_Width_Bytes']]).
lexEntry(pn_def,[symbol:'Rresp',num:_,syntax:['Rresp']]).
lexEntry(pn_def,[symbol:'Rlast',num:_,syntax:['Rlast']]).
lexEntry(pn_def,[symbol:'Addr_Width',num:_,syntax:['Addr_Width']]).
lexEntry(pn_def,[symbol:'Wdepth',num:_,syntax:['Wdepth']]).
lexEntry(pn_def,[symbol:'MaxrBursts',num:_,syntax:['MaxrBursts']]).
lexEntry(pn_def,[symbol:'MaxwBursts',num:_,syntax:['MaxwBursts']]).
lexEntry(pn_def,[symbol:'Awsnoop',num:_,syntax:['Awsnoop']]).
lexEntry(pn_def,[symbol:'Awbar',num:_,syntax:['Awbar']]).
lexEntry(pn_def,[symbol:'Awqos',num:_,syntax:['Awqos']]).
lexEntry(pn_def,[symbol:'Awregion',num:_,syntax:['Awregion']]).
lexEntry(pn_def,[symbol:'S_OK',num:_,syntax:['S_OK']]).
lexEntry(pn_def,[symbol:'Buser_Width',num:_,syntax:['Buser_Width']]).
lexEntry(pn_def,[symbol:'Buser',num:_,syntax:['Buser']]).
lexEntry(pn_def,[symbol:'Aruser_Width',num:_,syntax:['Aruser_Width']]).
lexEntry(pn_def,[symbol:'Aruser',num:_,syntax:['Aruser']]).
lexEntry(pn_def,[symbol:'MRMD',num:_,syntax:['MRMD']]).
lexEntry(pn_def,[symbol:'Arqos',num:_,syntax:['Arqos']]).
lexEntry(pn_def,[symbol:'Arregion',num:_,syntax:['Arregion']]).
lexEntry(pn_def,[symbol:'Ruser_Width',num:_,syntax:['Ruser_Width']]).
lexEntry(pn_def,[symbol:'Ruser',num:_,syntax:['Ruser']]).
lexEntry(pn_def,[symbol:'Data_width',num:_,syntax:['Data_width']]).
lexEntry(pn_def,[symbol:'Exmon_Width',num:_,syntax:['Exmon_Width']]).
lexEntry(pn_def,[symbol:'TID',num:_,syntax:['TID']]).
lexEntry(pn_def,[symbol:'Tready',num:_,syntax:['Tready']]).
lexEntry(pn_def,[symbol:'Tdest',num:_,syntax:['Tdest']]).
lexEntry(pn_def,[symbol:'Tdata',num:_,syntax:['Tdata']]).
lexEntry(pn_def,[symbol:'Tstrb',num:_,syntax:['Tstrb']]).
lexEntry(pn_def,[symbol:'Tlast',num:_,syntax:['Tlast']]).
lexEntry(pn_def,[symbol:'Tkeep',num:_,syntax:['Tkeep']]).
lexEntry(pn_def,[symbol:'Tuser',num:_,syntax:['Tuser']]).
lexEntry(pn_def,[symbol:'Dest_Width',num:_,syntax:['Dest_Width']]).
lexEntry(pn_def,[symbol:'User_Width',num:_,syntax:['User_Width']]).
lexEntry(pn_def,[symbol:'Arsnoop',num:_,syntax:['Arsnoop']]).
lexEntry(pn_def,[symbol:'Arbar',num:_,syntax:['Arbar']]).
lexEntry(pn_def,[symbol:'Resp[3]',num:_,syntax:['Resp[3]']]).
lexEntry(pn_def,[symbol:'Resp[2]',num:_,syntax:['Resp[2]']]).
lexEntry(pn_def,[symbol:'Rack',num:_,syntax:['Rack']]).
lexEntry(pn_def,[symbol:'Wack',num:_,syntax:['Wack']]).
lexEntry(pn_def,[symbol:'AW',num:_,syntax:['AW']]).
lexEntry(pn_def,[symbol:'Acvalid',num:_,syntax:['Acvalid']]).
lexEntry(pn_def,[symbol:'Crvalid',num:_,syntax:['Crvalid']]).
lexEntry(pn_def,[symbol:'Cdvalid',num:_,syntax:['Cdvalid']]).
lexEntry(pn_def,[symbol:'MaxcBursts',num:_,syntax:['MaxcBursts']]).
lexEntry(pn_def,[symbol:'Max_Barriers',num:_,syntax:['Max_Barriers']]).
lexEntry(pn_def,[symbol:'Acready',num:_,syntax:['Acready']]).
lexEntry(pn_def,[symbol:'Wuser',num:_,syntax:['Wuser']]).
lexEntry(pn_def,[symbol:'Wuser_Width',num:_,syntax:['Wuser_Width']]).
lexEntry(pn_def,[symbol:'ID_Width',num:_,syntax:['ID_Width']]).
lexEntry(pn_def,[symbol:'Acsnoop',num:_,syntax:['Acsnoop']]).
lexEntry(pn_def,[symbol:'Acprot',num:_,syntax:['Acprot']]).
lexEntry(pn_def,[symbol:'Crresp[2]',num:_,syntax:['Crresp[2]']]).
lexEntry(pn_def,[symbol:'Crresp',num:_,syntax:['Crresp']]).
lexEntry(pn_def,[symbol:'Crresp[4:0]',num:_,syntax:['Crresp[4:0]']]).
lexEntry(pn_def,[symbol:'Crready',num:_,syntax:['Crready']]).
lexEntry(pn_def,[symbol:'Cdlast',num:_,syntax:['Cdlast']]).
lexEntry(pn_def,[symbol:'Cddata',num:_,syntax:['Cddata']]).
lexEntry(pn_def,[symbol:'Cdready',num:_,syntax:['Cdready']]).
lexEntry(pn_def,[symbol:'Acaddr',num:_,syntax:['Acaddr']]).
lexEntry(pn_def,[symbol:'CD_Data_Width',num:_,syntax:['CD_Data_Width']]).
lexEntry(pn_def,[symbol:'Addr',num:_,syntax:['Addr']]).
lexEntry(pn_def,[symbol:'Mcmd',num:_,syntax:['Mcmd']]).
lexEntry(pn_def,[symbol:'AtomicLength',num:_,syntax:['AtomicLength']]).
lexEntry(pn_def,[symbol:'Atomiclength_Wdth',num:_,syntax:['Atomiclength_Wdth']]).
lexEntry(pn_def,[symbol:'BCST',num:_,syntax:['BCST']]).
lexEntry(pn_def,[symbol:'MBlockHeight',num:_,syntax:['MBlockHeight']]).
lexEntry(pn_def,[symbol:'BlockHeight_Wdth',num:_,syntax:['BlockHeight_Wdth']]).
lexEntry(pn_def,[symbol:'BlockStride',num:_,syntax:['BlockStride']]).
lexEntry(pn_def,[symbol:'Broadcast_Enable',num:_,syntax:['Broadcast_Enable']]).
lexEntry(pn_def,[symbol:'Burst_Aligned',num:_,syntax:['Burst_Aligned']]).
lexEntry(pn_def,[symbol:'BurstLength',num:_,syntax:['BurstLength']]).
lexEntry(pn_def,[symbol:'MBlockStride[0]',num:_,syntax:['MBlockStride[0]']]).
lexEntry(pn_def,[symbol:'MBlockStride[1:0]',num:_,syntax:['MBlockStride[1:0]']]).
lexEntry(pn_def,[symbol:'MBlockStride[2:0]',num:_,syntax:['MBlockStride[2:0]']]).
lexEntry(pn_def,[symbol:'MBlockStride[3:0]',num:_,syntax:['MBlockStride[3:0]']]).
lexEntry(pn_def,[symbol:'Maddr[0]',num:_,syntax:['Maddr[0]']]).
lexEntry(pn_def,[symbol:'Maddr[1:0]',num:_,syntax:['Maddr[1:0]']]).
lexEntry(pn_def,[symbol:'Maddr[2:0]',num:_,syntax:['Maddr[2:0]']]).
lexEntry(pn_def,[symbol:'Maddr[3:0]',num:_,syntax:['Maddr[3:0]']]).
lexEntry(pn_def,[symbol:'BurstLength_Wdth',num:_,syntax:['BurstLength_Wdth']]).
lexEntry(pn_def,[symbol:'BurstPrecise',num:_,syntax:['BurstPrecise']]).
lexEntry(pn_def,[symbol:'BurstSeq',num:_,syntax:['BurstSeq']]).
lexEntry(pn_def,[symbol:'BurstSeq_Blck_Enable',num:_,syntax:['BurstSeq_Blck_Enable']]).
lexEntry(pn_def,[symbol:'BurstSeq_Dflt1_Enable',num:_,syntax:['BurstSeq_Dflt1_Enable']]).
lexEntry(pn_def,[symbol:'BurstSeq_Dflt2_Enable',num:_,syntax:['BurstSeq_Dflt2_Enable']]).
lexEntry(pn_def,[symbol:'BurstSeq_Enable',num:_,syntax:['BurstSeq_Enable']]).
lexEntry(pn_def,[symbol:'BurstSeq_Incr_Enable',num:_,syntax:['BurstSeq_Incr_Enable']]).
lexEntry(pn_def,[symbol:'BurstSeq_Strm_Enable',num:_,syntax:['BurstSeq_Strm_Enable']]).
lexEntry(pn_def,[symbol:'BurstSeq_Unkn_Enable',num:_,syntax:['BurstSeq_Unkn_Enable']]).
lexEntry(pn_def,[symbol:'BurstSeq_Wrap_Enable',num:_,syntax:['BurstSeq_Wrap_Enable']]).
lexEntry(pn_def,[symbol:'BurstSeq_XOR_Enable',num:_,syntax:['BurstSeq_XOR_Enable']]).
lexEntry(pn_def,[symbol:'BurstSingleReq',num:_,syntax:['BurstSingleReq']]).
lexEntry(pn_def,[symbol:'ByteEn',num:_,syntax:['ByteEn']]).
lexEntry(pn_def,[symbol:'CmdAccept',num:_,syntax:['CmdAccept']]).
lexEntry(pn_def,[symbol:'ConnectCap',num:_,syntax:['ConnectCap']]).
lexEntry(pn_def,[symbol:'Connection',num:_,syntax:['Connection']]).
lexEntry(pn_def,[symbol:'Control',num:_,syntax:['Control']]).
lexEntry(pn_def,[symbol:'ControlBusy',num:_,syntax:['ControlBusy']]).
lexEntry(pn_def,[symbol:'ControlWr',num:_,syntax:['ControlWr']]).
lexEntry(pn_def,[symbol:'Corresponding_Exact',num:_,syntax:['Corresponding_Exact']]).
lexEntry(pn_def,[symbol:'DataAccept',num:_,syntax:['DataAccept']]).
lexEntry(pn_def,[symbol:'Datahandshake',num:_,syntax:['Datahandshake']]).
lexEntry(pn_def,[symbol:'DataLast',num:_,syntax:['DataLast']]).
lexEntry(pn_def,[symbol:'DataRowLast',num:_,syntax:['DataRowLast']]).
lexEntry(pn_def,[symbol:'Data_Wdth',num:_,syntax:['Data_Wdth']]).
lexEntry(pn_def,[symbol:'DLFT1',num:_,syntax:['DLFT1']]).
lexEntry(pn_def,[symbol:'DLFT2',num:_,syntax:['DLFT2']]).
lexEntry(pn_def,[symbol:'DFLT2',num:_,syntax:['DFLT2']]).
lexEntry(pn_def,[symbol:'SThreadBusy_Exact',num:_,syntax:['SThreadBusy_Exact']]).
lexEntry(pn_def,[symbol:'SThreadBusy_ExactCan',num:_,syntax:['SThreadBusy_ExactCan']]).
lexEntry(pn_def,[symbol:'MThreadBusy_Exact',num:_,syntax:['MThreadBusy_Exact']]).
lexEntry(pn_def,[symbol:'SDataThreadBusy_Exact',num:_,syntax:['SDataThreadBusy_Exact']]).
lexEntry(pn_def,[symbol:'Force_Aligned',num:_,syntax:['Force_Aligned']]).
lexEntry(pn_def,[symbol:'IDLE',num:_,syntax:['IDLE']]).
lexEntry(pn_def,[symbol:'INCR',num:_,syntax:['INCR']]).
lexEntry(pn_def,[symbol:'Jtag_Enable',num:_,syntax:['Jtag_Enable']]).
lexEntry(pn_def,[symbol:'Jtagtrst_Enable',num:_,syntax:['Jtagtrst_Enable']]).
lexEntry(pn_def,[symbol:'MAddr',num:_,syntax:['MAddr']]).
lexEntry(pn_def,[symbol:'MAddrSpace',num:_,syntax:['MAddrSpace']]).
lexEntry(pn_def,[symbol:'MAtomicLength',num:_,syntax:['MAtomicLength']]).
lexEntry(pn_def,[symbol:'MBlockHeight',num:_,syntax:['MBlockHeight']]).
lexEntry(pn_def,[symbol:'MBurstPrecise',num:_,syntax:['MBurstPrecise']]).
lexEntry(pn_def,[symbol:'MBlockStride',num:_,syntax:['MBlockStride']]).
lexEntry(pn_def,[symbol:'MBurstLength',num:_,syntax:['MBurstLength']]).
lexEntry(pn_def,[symbol:'MBurstSeq',num:_,syntax:['MBurstSeq']]).
lexEntry(pn_def,[symbol:'MBurstSingleReq',num:_,syntax:['MBurstSingleReq']]).
lexEntry(pn_def,[symbol:'MByteEn',num:_,syntax:['MByteEn']]).
lexEntry(pn_def,[symbol:'M_CON',num:_,syntax:['M_CON']]).
lexEntry(pn_def,[symbol:'MConnect',num:_,syntax:['MConnect']]).
lexEntry(pn_def,[symbol:'MConnID',num:_,syntax:['MConnID']]).
lexEntry(pn_def,[symbol:'MData',num:_,syntax:['MData']]).
lexEntry(pn_def,[symbol:'MDataByteEn',num:_,syntax:['MDataByteEn']]).
lexEntry(pn_def,[symbol:'MDataInfo',num:_,syntax:['MDataInfo']]).
lexEntry(pn_def,[symbol:'MDataInfoByte_Wdth',num:_,syntax:['MDataInfoByte_Wdth']]).
lexEntry(pn_def,[symbol:'MDataLast',num:_,syntax:['MDataLast']]).
lexEntry(pn_def,[symbol:'MDataRowLast',num:_,syntax:['MDataRowLast']]).
lexEntry(pn_def,[symbol:'MDataTagID',num:_,syntax:['MDataTagID']]).
lexEntry(pn_def,[symbol:'MDataThreadID',num:_,syntax:['MDataThreadID']]).
lexEntry(pn_def,[symbol:'MDataValid',num:_,syntax:['MDataValid']]).
lexEntry(pn_def,[symbol:'M_DISC',num:_,syntax:['M_DISC']]).
lexEntry(pn_def,[symbol:'M_OFF',num:_,syntax:['M_OFF']]).
lexEntry(pn_def,[symbol:'MError',num:_,syntax:['MError']]).
lexEntry(pn_def,[symbol:'MReqInfo',num:_,syntax:['MReqInfo']]).
lexEntry(pn_def,[symbol:'MReqLast',num:_,syntax:['MReqLast']]).
lexEntry(pn_def,[symbol:'MReqRowLast',num:_,syntax:['MReqRowLast']]).
lexEntry(pn_def,[symbol:'MReset',num:_,syntax:['MReset']]).
lexEntry(pn_def,[symbol:'MReset_n',num:_,syntax:['MReset_n']]).
lexEntry(pn_def,[symbol:'Reset',num:_,syntax:['Reset']]).
lexEntry(pn_def,[symbol:'MRespAccept',num:_,syntax:['MRespAccept']]).
lexEntry(pn_def,[symbol:'MTagID',num:_,syntax:['MTagID']]).
lexEntry(pn_def,[symbol:'MTagInOrder',num:_,syntax:['MTagInOrder']]).
lexEntry(pn_def,[symbol:'MThreadBusy',num:_,syntax:['MThreadBusy']]).
lexEntry(pn_def,[symbol:'MThreadID',num:_,syntax:['MThreadID']]).
lexEntry(pn_def,[symbol:'SThreadBusy_Pipelined',num:_,syntax:['SThreadBusy_Pipelined']]).
lexEntry(pn_def,[symbol:'MThreadBusy_Pipelined',num:_,syntax:['MThreadBusy_Pipelined']]).
lexEntry(pn_def,[symbol:'SDataThreadBusy_Pipelined',num:_,syntax:['
    SDataThreadBusy_Pipelined']]).
lexEntry(pn_def,[symbol:'RD',num:_,syntax:['RD']]).
lexEntry(pn_def,[symbol:'RDEX',num:_,syntax:['RDEX']]).
lexEntry(pn_def,[symbol:'RDL',num:_,syntax:['RDL']]).
lexEntry(pn_def,[symbol:'RDLwrc_Enable',num:_,syntax:['RDLwrc_Enable']]).
lexEntry(pn_def,[symbol:'Read_Enable',num:_,syntax:['Read_Enable']]).
lexEntry(pn_def,[symbol:'ReadEx',num:_,syntax:['ReadEx']]).
lexEntry(pn_def,[symbol:'ReadEx_Enable',num:_,syntax:['ReadEx_Enable']]).
lexEntry(pn_def,[symbol:'ReqData_Together',num:_,syntax:['ReqData_Together']]).
lexEntry(pn_def,[symbol:'ReqLast',num:_,syntax:['ReqLast']]).
lexEntry(pn_def,[symbol:'ReqRowLast',num:_,syntax:['ReqRowLast']]).
lexEntry(pn_def,[symbol:'RESP',num:_,syntax:['RESP']]).
lexEntry(pn_def,[symbol:'RespAccept',num:_,syntax:['RespAccept']]).
lexEntry(pn_def,[symbol:'RespInfo',num:_,syntax:['RespInfo']]).
lexEntry(pn_def,[symbol:'Res_ast',num:_,syntax:['Res_ast']]).
lexEntry(pn_def,[symbol:'RespRowLast',num:_,syntax:['RespRowLast']]).
lexEntry(pn_def,[symbol:'RiteNonPost_Enable',num:_,syntax:['RiteNonPost_Enable']]).
lexEntry(pn_def,[symbol:'SCmdAccept',num:_,syntax:['SCmdAccept']]).
lexEntry(pn_def,[symbol:'S_CON',num:_,syntax:['S_CON']]).
lexEntry(pn_def,[symbol:'SConnect',num:_,syntax:['SConnect']]).
lexEntry(pn_def,[symbol:'SData',num:_,syntax:['SData']]).
lexEntry(pn_def,[symbol:'SDataAccept',num:_,syntax:['SDataAccept']]).
lexEntry(pn_def,[symbol:'SDataInfo',num:_,syntax:['SDataInfo']]).
lexEntry(pn_def,[symbol:'SDataInfoByte_Wdth',num:_,syntax:['SDataInfoByte_Wdth']]).
lexEntry(pn_def,[symbol:'SDataInfo_Wdth',num:_,syntax:['SDataInfo_Wdth']]).
lexEntry(pn_def,[symbol:'SDataThreadBusy',num:_,syntax:['SDataThreadBusy']]).
lexEntry(pn_def,[symbol:'SError',num:_,syntax:['SError']]).
lexEntry(pn_def,[symbol:'SInterrupt',num:_,syntax:['SInterrupt']]).
lexEntry(pn_def,[symbol:'SReset',num:_,syntax:['SReset']]).
lexEntry(pn_def,[symbol:'SReset_n',num:_,syntax:['SReset_n']]).
lexEntry(pn_def,[symbol:'SResp',num:_,syntax:['SResp']]).
lexEntry(pn_def,[symbol:'SRespInfo',num:_,syntax:['SRespInfo']]).
lexEntry(pn_def,[symbol:'SRes_ast',num:_,syntax:['SRes_ast']]).
lexEntry(pn_def,[symbol:'SRespRowLast',num:_,syntax:['SRespRowLast']]).
lexEntry(pn_def,[symbol:'SRMD',num:_,syntax:['SRMD']]).
lexEntry(pn_def,[symbol:'STagID',num:_,syntax:['STagID']]).
lexEntry(pn_def,[symbol:'STagInOrder',num:_,syntax:['STagInOrder']]).
lexEntry(pn_def,[symbol:'Status',num:_,syntax:['Status']]).
lexEntry(pn_def,[symbol:'StatusBusy',num:_,syntax:['StatusBusy']]).
lexEntry(pn_def,[symbol:'StatusRd',num:_,syntax:['StatusRd']]).
lexEntry(pn_def,[symbol:'SThreadBusy',num:_,syntax:['SThreadBusy']]).
lexEntry(pn_def,[symbol:'SThreadID',num:_,syntax:['SThreadID']]).
lexEntry(pn_def,[symbol:'STRM',num:_,syntax:['STRM']]).
lexEntry(pn_def,[symbol:'SWait',num:_,syntax:['SWait']]).
lexEntry(pn_def,[symbol:'TagInOrder',num:_,syntax:['TagInOrder']]).
lexEntry(pn_def,[symbol:'UNKN',num:_,syntax:['UNKN']]).
lexEntry(pn_def,[symbol:'MDataInfo_Wdth',num:_,syntax:['MDataInfo_Wdth']]).
lexEntry(pn_def,[symbol:'MDataInfoByte_Wdth',num:_,syntax:['MDataInfoByte_Wdth']]).
lexEntry(pn_def,[symbol:'WR',num:_,syntax:['WR']]).
lexEntry(pn_def,[symbol:'WRC',num:_,syntax:['WRC']]).
lexEntry(pn_def,[symbol:'Write_Enable',num:_,syntax:['Write_Enable']]).
lexEntry(pn_def,[symbol:'WriteNonPost_Enable',num:_,syntax:['WriteNonPost_Enable']]).
lexEntry(pn_def,[symbol:'WRNP',num:_,syntax:['WRNP']]).
lexEntry(pn_def,[symbol:'MRespRowLast',num:_,syntax:['MRespRowLast']]).
lexEntry(pn_def,[symbol:'XOR',num:_,syntax:['XOR']]).
lexEntry(pn_def,[symbol:'Barrier',num:_,syntax:['Barrier']]).
lexEntry(pn_def,[symbol:'Data_Width',num:_,syntax:['Data','Width']]).
lexEntry(pn_def,[symbol:'AC_payload',num:_,syntax:['AC',payload]]).
lexEntry(pn_def,[symbol:'CD_payload',num:_,syntax:['CD',payload]]).
lexEntry(pn_def,[symbol:'CR_payload',num:_,syntax:['CR',payload]]).
lexEntry(pn_def,[symbol:'TUSER_payload',num:_,syntax:['TUSER',payload]]).
lexEntry(pn_def,[symbol:'RData_valid_byte_lanes',num:_,syntax:['RData',valid,byte,lanes
    ]]).
lexEntry(pn_def,[symbol:'WData_valid_byte_lanes',num:_,syntax:['WData',valid,byte,lanes
    ]]).

/*==========================================================================
                              Temporal Nouns
==========================================================================*/
lexEntry(tnoun,[symbol:reset,num:sg,syntax:[reset]]).
lexEntry(tnoun,[symbol:accesse,num:pl,syntax:[accesses]]).
lexEntry(tnoun,[symbol:request,num:sg,syntax:[request]]).
lexEntry(tnoun,[symbol:request,num:pl,syntax:[requests]]).
lexEntry(tnoun,[symbol:response,num:sg,syntax:[response]]).
lexEntry(tnoun,[symbol:ack,num:sg,syntax:[acknowledgement]]).
lexEntry(tnoun,[symbol:transaction,num:sg,syntax:[transaction]]).
lexEntry(tnoun,[symbol:cycle,num:sg,syntax:[cycle]]).
lexEntry(tnoun,[symbol:sequence,num:sg,syntax:[sequence]]).
lexEntry(tnoun,[symbol:transaction,num:pl,syntax:[transactions]]).
lexEntry(tnoun,[symbol:cycle,num:pl,syntax:[cycles]]).
lexEntry(tnoun,[symbol:cycle,num:pl,syntax:[consecutive,cycles]]).
lexEntry(tnoun,[symbol:sequence,num:pl,syntax:[sequences]]).
lexEntry(tnoun,[symbol:phase,num:sg,syntax:[phase]]).
lexEntry(tnoun,[symbol:ocp_reset,num:sg,syntax:['OCP',reset]]).
lexEntry(tnoun,[symbol:ocp_transaction,num:sg,syntax:['OCP',transaction]]).
lexEntry(tnoun,[symbol:ocp_clock,num:sg,syntax:['OCP',clock]]).
lexEntry(tnoun,[symbol:clock_cycle,num:sg,syntax:[clock,cycle]]).
lexEntry(tnoun,[symbol:clock_cycle,num:pl,syntax:[clock,cycles]]).
lexEntry(tnoun,[symbol:clock_signal,num:sg,syntax:[clock,signal]]).
lexEntry(tnoun,[symbol:request_phase,num:sg,syntax:[request,phase]]).
lexEntry(tnoun,[symbol:response_phase,num:sg,syntax:[response,phase]]).
lexEntry(tnoun,[symbol:datahandshake_phase,num:sg,syntax:[datahandshake,phase]]).
lexEntry(tnoun,[symbol:data_phase,num:sg,syntax:[data,phase]]).
lexEntry(tnoun,[symbol:response_datahandshake,num:sg,syntax:[response,datahandshake]]).
lexEntry(tnoun,[symbol:transfer,num:pl,syntax:[transfers]]).
lexEntry(tnoun,[symbol:data_transfer,num:pl,syntax:[data,transfers]]).
lexEntry(tnoun,[symbol:burst_sequence,num:pl,syntax:[burst,sequences]]).
lexEntry(tnoun,[symbol:transaction_sequence,num:pl,syntax:[transaction,sequences]]).
lexEntry(tnoun,[symbol:burst_request,num:sg,syntax:[burst,request]]).
lexEntry(tnoun,[symbol:wrc_request,num:sg,syntax:['WRC',request]]).
lexEntry(tnoun,[symbol:dvm_response,num:sg,syntax:['DVM',response]]).
lexEntry(tnoun,[symbol:aw_handshake,num:sg,syntax:['AW',handshake]]).
lexEntry(tnoun,[symbol:rlast_handshake,num:sg,syntax:['Rlast',handshake]]).
lexEntry(tnoun,[symbol:burst,num:sg,syntax:[burst]]).
lexEntry(tnoun,[symbol:burst,num:pl,syntax:[bursts]]).
lexEntry(tnoun,[symbol:dflt2_bursts,num:pl,syntax:['DFLT2',bursts]]).
lexEntry(tnoun,[symbol:srmd_burst,num:sg,syntax:['SRMD',burst]]).
lexEntry(tnoun,[symbol:mrmd_burst,num:sg,syntax:['MRMD',burst]]).
lexEntry(tnoun,[symbol:incr_burst,num:sg,syntax:['INCR',burst]]).
lexEntry(tnoun,[symbol:incr_burst,num:sg,syntax:['WRAP',burst]]).
lexEntry(tnoun,[symbol:precise_burst,num:pl,syntax:[precise,bursts]]).
lexEntry(tnoun,[symbol:evict_transaction,num:pl,syntax:['Evict',transactions]]).
lexEntry(tnoun,[symbol:dvm_complete_transaction,num:pl,syntax:['DVM',complete,
    transactions]]).
lexEntry(tnoun,[symbol:dvm_read_transaction,num:pl,syntax:['DVM',read,transactions]]).
lexEntry(tnoun,[symbol:writenosnoop_transaction,num:pl,syntax:['WriteNoSnoop',
    transactions]]).
/*=============================================================================
                              Number
=============================================================================*/
lexEntry(number,[symbol:'0',syntax:[zero]]).
lexEntry(number,[symbol:'0',syntax:[zeros]]).
lexEntry(number,[symbol:'0',syntax:['0']]).
lexEntry(number,[symbol:'1',syntax:['1']]).
lexEntry(number,[symbol:'1',syntax:['1s']]).
lexEntry(number,[symbol:'2',syntax:['2']]).
lexEntry(number,[symbol:'3',syntax:['3']]).
lexEntry(number,[symbol:'4',syntax:['4']]).
lexEntry(number,[symbol:'5',syntax:['5']]).
lexEntry(number,[symbol:'6',syntax:['6']]).
lexEntry(number,[symbol:'7',syntax:['7']]).
lexEntry(number,[symbol:'8',syntax:['8']]).
lexEntry(number,[symbol:'9',syntax:['9']]).
lexEntry(number,[symbol:'10',syntax:['10']]).
lexEntry(number,[symbol:'16',syntax:['16']]).
lexEntry(number,[symbol:'32',syntax:['32']]).
lexEntry(number,[symbol:'64',syntax:['64']]).
lexEntry(number,[symbol:'128',syntax:['128']]).
lexEntry(number,[symbol:'256',syntax:['256']]).
lexEntry(number,[symbol:'512',syntax:['512']]).
lexEntry(number,[symbol:'1024',syntax:['1024']]).
lexEntry(number,[symbol:'16',syntax:['MaxWaits']]).
lexEntry(number,[symbol:'1',syntax:[one]]).
lexEntry(number,[symbol:'2',syntax:[two]]).
lexEntry(number,[symbol:'3',syntax:[three]]).
lexEntry(number,[symbol:'4',syntax:[four]]).
lexEntry(number,[symbol:'5',syntax:[five]]).
lexEntry(number,[symbol:'6',syntax:[six]]).
lexEntry(number,[symbol:'7',syntax:[seven]]).
lexEntry(number,[symbol:'8',syntax:[eight]]).
lexEntry(number,[symbol:'9',syntax:[nine]]).
lexEntry(number,[symbol:'10',syntax:[ten]]).
lexEntry(number,[symbol:'5 b000x0',syntax:['5 b000x0']]).
lexEntry(number,[symbol:'2 b11',syntax:['1 b1']]).
lexEntry(number,[symbol:'2 b00',syntax:['1 b0']]).
lexEntry(number,[symbol:'2 b11',syntax:['2 b11']]).
lexEntry(number,[symbol:'2 b00',syntax:['2 b00']]).
lexEntry(number,[symbol:'2 b0000',syntax:['2 b0000']]).
lexEntry(number,[symbol:'0001',syntax:['0001']]).
lexEntry(number,[symbol:'0010',syntax:['0010']]).
lexEntry(number,[symbol:'0100',syntax:['0100']]).
lexEntry(number,[symbol:'1000',syntax:['1000']]).
lexEntry(number,[symbol:'0011',syntax:['0011']]).
lexEntry(number,[symbol:'1100',syntax:['1100']]).
lexEntry(number,[symbol:'1111',syntax:['1111']]).
lexEntry(number,[symbol:'0000',syntax:['0000']]).
lexEntry(number,[symbol:'00000001',syntax:['00000001']]).
lexEntry(number,[symbol:'00000010',syntax:['00000010']]).
lexEntry(number,[symbol:'00000100',syntax:['00000100']]).
lexEntry(number,[symbol:'00001000',syntax:['00001000']]).
lexEntry(number,[symbol:'00010000',syntax:['00010000']]).
lexEntry(number,[symbol:'00100000',syntax:['00100000']]).
lexEntry(number,[symbol:'01000000',syntax:['01000000']]).
lexEntry(number,[symbol:'10000000',syntax:['10000000']]).
lexEntry(number,[symbol:'00000011',syntax:['00000011']]).
lexEntry(number,[symbol:'00001100',syntax:['00001100']]).
lexEntry(number,[symbol:'00110000',syntax:['00110000']]).
lexEntry(number,[symbol:'11000000',syntax:['11000000']]).
lexEntry(number,[symbol:'00001111',syntax:['00001111']]).
lexEntry(number,[symbol:'11110000',syntax:['11110000']]).
lexEntry(number,[symbol:'11111111',syntax:['11111111']]).
lexEntry(number,[symbol:'00000000',syntax:['00000000']]).
lexEntry(number,[symbol:'0000000000000001',syntax:['0000000000000001']]).
lexEntry(number,[symbol:'0000000000000010',syntax:['0000000000000010']]).
lexEntry(number,[symbol:'0000000000000100',syntax:['0000000000000100']]).
lexEntry(number,[symbol:'0000000000001000',syntax:['0000000000001000']]).
lexEntry(number,[symbol:'0000000000010000',syntax:['0000000000010000']]).
lexEntry(number,[symbol:'0000000000100000',syntax:['0000000000100000']]).
lexEntry(number,[symbol:'0000000001000000',syntax:['0000000001000000']]).
lexEntry(number,[symbol:'0000000010000000',syntax:['0000000010000000']]).
lexEntry(number,[symbol:'0000000100000000',syntax:['0000000100000000']]).
lexEntry(number,[symbol:'0000001000000000',syntax:['0000001000000000']]).
lexEntry(number,[symbol:'0000010000000000',syntax:['0000010000000000']]).
lexEntry(number,[symbol:'0000100000000000',syntax:['0000100000000000']]).
lexEntry(number,[symbol:'0001000000000000',syntax:['0001000000000000']]).
lexEntry(number,[symbol:'0010000000000000',syntax:['0010000000000000']]).
lexEntry(number,[symbol:'0100000000000000',syntax:['0100000000000000']]).
lexEntry(number,[symbol:'1000000000000000',syntax:['1000000000000000']]).
lexEntry(number,[symbol:'0000000000000011',syntax:['0000000000000011']]).
lexEntry(number,[symbol:'0000000000001100',syntax:['0000000000001100']]).
lexEntry(number,[symbol:'0000000000110000',syntax:['0000000000110000']]).
lexEntry(number,[symbol:'0000000011000000',syntax:['0000000011000000']]).
lexEntry(number,[symbol:'0000001100000000',syntax:['0000001100000000']]).
lexEntry(number,[symbol:'0011000000000000',syntax:['0011000000000000']]).
lexEntry(number,[symbol:'1100000000000000',syntax:['1100000000000000']]).
lexEntry(number,[symbol:'0000000000001111',syntax:['0000000000001111']]).
lexEntry(number,[symbol:'0000000011110000',syntax:['0000000011110000']]).
lexEntry(number,[symbol:'0000111100000000',syntax:['0000111100000000']]).
lexEntry(number,[symbol:'1111000000000000',syntax:['1111000000000000']]).
lexEntry(number,[symbol:'0000000011111111',syntax:['0000000011111111']]).
lexEntry(number,[symbol:'1111111100000000',syntax:['1111111100000000']]).
lexEntry(number,[symbol:'1111111111111111',syntax:['1111111111111111']]).
lexEntry(number,[symbol:'0000000000000000',syntax:['0000000000000000']]).

/*=========================================================================
                         Intransitive verbs
=========================================================================*/

lexEntry(iv,[syntax:accept]).
lexEntry(iv,[syntax:exist]).
lexEntry(iv,[syntax:go]).
lexEntry(iv,[syntax:hold]).
lexEntry(iv,[syntax:keep]).
lexEntry(iv,[syntax:begin]).
lexEntry(iv,[syntax:set]).
lexEntry(iv,[syntax:take]).
lexEntry(iv,[syntax:change]).
lexEntry(iv,[syntax:match]).
lexEntry(iv,[syntax:remain]).
lexEntry(iv,[syntax:assert]).
lexEntry(iv,[syntax:deassert]).
lexEntry(iv,[syntax:be]).
lexEntry(iv,[syntax:permit]).
lexEntry(iv,[syntax:change]).
lexEntry(iv,[syntax:drive]).
lexEntry(iv,[syntax:stay]).
lexEntry(iv,[syntax:stall]).
lexEntry(iv,[syntax:occur]).
lexEntry(iv,[syntax:enable]).
lexEntry(iv,[syntax:fail]).
lexEntry(iv,[syntax:become]).
/*=========================================================================
                         Transitive verbs
=========================================================================*/
lexEntry(tv,[syntax:be]).
lexEntry(tv,[syntax:accept]).
lexEntry(tv,[syntax:allow]).
lexEntry(tv,[syntax:assign]).
lexEntry(tv,[syntax:has]).
lexEntry(tv,[syntax:have]).
lexEntry(tv,[syntax:give]).
lexEntry(tv,[syntax:go]).
lexEntry(tv,[syntax:hold]).
lexEntry(tv,[syntax:permit]).
lexEntry(tv,[syntax:cross]).
lexEntry(tv,[syntax:issue]).
lexEntry(tv,[syntax:complete]).
lexEntry(tv,[syntax:change]).
lexEntry(tv,[syntax:limit]).
lexEntry(tv,[syntax:sample]).
lexEntry(tv,[syntax:produce]).
lexEntry(tv,[syntax:match]).
lexEntry(tv,[syntax:interleave]).
lexEntry(tv,[syntax:restrict]).
lexEntry(tv,[syntax:set]).
lexEntry(tv,[syntax:transition]).
/*=========================================================================
                         Ditransitive verbs
=========================================================================*/
lexEntry(dtv,[syntax:allow]).
lexEntry(dtv,[syntax:assign]).
lexEntry(dtv,[syntax:permit]).
lexEntry(dtv,[syntax:get]).
lexEntry(dtv,[syntax:give]).
/*=========================================================================
                         Adjectives
=========================================================================*/
lexEntry(adj,[symbol:associated,syntax:[associated]]).
lexEntry(adj,[symbol:false,syntax:[false]]).
lexEntry(adj,[symbol:given,syntax:[given]]).
lexEntry(adj,[symbol:permitted,syntax:[permitted]]).
lexEntry(adj,[symbol:read,syntax:[read]]).
lexEntry(adj,[symbol:invalid,syntax:[invalid]]).
lexEntry(adj,[symbol:true,syntax:[true]]).
lexEntry(adj,[symbol:valid,syntax:[valid]]).
lexEntry(adj,[symbol:accepted,syntax:[accepted]]).
lexEntry(adj,[symbol:active,syntax:[active]]).
lexEntry(adj,[symbol:allowed,syntax:[allowed]]).
lexEntry(adj,[symbol:locked,syntax:[locked]]).
lexEntry(adj,[symbol:write,syntax:[write]]).
lexEntry(adj,[symbol:unlocked,syntax:[unlocked]]).
lexEntry(adj,[symbol:full,syntax:[full]]).
lexEntry(adj,[symbol:reserved,syntax:[reserved]]).
lexEntry(adj,[symbol:read,syntax:[read]]).
lexEntry(adj,[symbol:exclusive,syntax:[exclusive]]).
lexEntry(adj,[symbol:additional,syntax:[additional]]).
lexEntry(adj,[symbol:asserted,syntax:[asserted]]).
lexEntry(adj,[symbol:outsting,syntax:[outsting]]).
lexEntry(adj,[symbol:same,syntax:[same]]).
lexEntry(adj,[symbol:shareable,syntax:[shareable]]).
lexEntry(adj,[symbol:corresponding,syntax:[corresponding]]).
lexEntry(adj,[symbol:required,syntax:[required]]).
lexEntry(adj,[symbol:aligned,syntax:[aligned]]).
lexEntry(adj,[symbol:outer,syntax:[outer]]).
lexEntry(adj,[symbol:inner,syntax:[inner]]).
lexEntry(adj,[symbol:'4kb',syntax:['4KB']]).
lexEntry(adj,[symbol:inclusive,syntax:[inclusive]]).
lexEntry(adj,[symbol:single,syntax:[single]]).
lexEntry(adj,[symbol:high,syntax:[high]]).
lexEntry(adj,[symbol:low,syntax:[low]]).
lexEntry(adj,[symbol:non_finshed,syntax:[non,finshed]]).
lexEntry(adj,[symbol:disconnect,syntax:[disconnect]]).
lexEntry(adj,[symbol:connected,syntax:[connected]]).
lexEntry(adj,[symbol:disabled,syntax:[disabled]]).
lexEntry(adj,[symbol:non_blck,syntax:[non,blck]]).
lexEntry(adj,[symbol:rising,syntax:[rising]]).
lexEntry(adj,[symbol:equal,syntax:[equal]]).
lexEntry(adj,[symbol:different,syntax:[different]]).
lexEntry(adj,[symbol:provided,syntax:[provided]]).
lexEntry(adj,[symbol:inactive,syntax:[inactive]]).
lexEntry(adj,[symbol:enabled,syntax:[enabled]]).
lexEntry(adj,[symbol:non_enabled,syntax:[non,enabled]]).
lexEntry(adj,[symbol:block,syntax:[block]]).
lexEntry(adj,[symbol:absent,syntax:[absent]]).
lexEntry(adj,[symbol:legal,syntax:[legal]]).
lexEntry(adj,[symbol:illegal,syntax:[illegal]]).
lexEntry(adj,[symbol:active,syntax:[active]]).
lexEntry(adj,[symbol:busy,syntax:[busy]]).
lexEntry(adj,[symbol:unknown,syntax:[unknown]]).
lexEntry(adj,[symbol:normal,syntax:[normal]]).
lexEntry(adj,[symbol:outstanding,syntax:[outstanding]]).
lexEntry(adj,[symbol:supported,syntax:[supported]]).
lexEntry(adj,[symbol:tagged,syntax:[tagged]]).
lexEntry(adj,[symbol:unlocking,syntax:[unlocking]]).
lexEntry(adj,[symbol:exokay,syntax:['EXOKAY']]).
lexEntry(adj,[symbol:passdirty,syntax:['PassDirty']]).
lexEntry(adj,[symbol:isshared,syntax:['IsShared']]).
lexEntry(adj,[symbol:cleanunique,syntax:['CleanUnique']]).
lexEntry(adj,[symbol:cleanshared,syntax:['CleanShared']]).
lexEntry(adj,[symbol:cleaninvalid,syntax:['CleanInvalid']]).
lexEntry(adj,[symbol:makeunique,syntax:['MakeUnique']]).
lexEntry(adj,[symbol:makeinvalid,syntax:['MakeInvalid']]).
lexEntry(adj,[symbol:make,syntax:['Make']]).
lexEntry(adj,[symbol:clean,syntax:['Clean']]).
lexEntry(adj,[symbol:readunique,syntax:['ReadUnique']]).
lexEntry(adj,[symbol:readshared,syntax:['ReadShared']]).
lexEntry(adj,[symbol:readonce,syntax:['ReadOnce']]).
lexEntry(adj,[symbol:readnotshareddirty,syntax:['ReadNotSharedDirty']]).
lexEntry(adj,[symbol:readnosnoop,syntax:['ReadNoSnoop']]).
lexEntry(adj,[symbol:readclean,syntax:['ReadClean']]).
lexEntry(adj,[symbol:writeunique,syntax:['WriteUnique']]).
lexEntry(adj,[symbol:writenosnoop,syntax:['WriteNoSnoop']]).
lexEntry(adj,[symbol:writelineunique,syntax:['WriteLineUnique']]).
lexEntry(adj,[symbol:writeclean,syntax:['WriteClean']]).
lexEntry(adj,[symbol:writeback,syntax:['WriteBack']]).
lexEntry(adj,[symbol:fail,syntax:['FAIL']]).
lexEntry(adj,[symbol:blck,syntax:['BLCK']]).
lexEntry(adj,[symbol:non_blck,syntax:[non,'BLCK']]).
lexEntry(adj,[symbol:barrier,syntax:[barrier]]).
lexEntry(adj,[symbol:non_barrier,syntax:[non,barrier]]).
lexEntry(adj,[symbol:non_inorder,syntax:[non,inorder]]).
/*=====================================================================
                         Adjective (Stable)
=====================================================================*/
lexEntry(adj_Stable,[syntax:[stable]]).
lexEntry(adj_Stable,[syntax:[steady]]).
lexEntry(adj_Stable,[syntax:[constant]]).
/*=====================================================================
                         Adverbs
=====================================================================*/
lexEntry(adv,[syntax:[exactly]]).
lexEntry(adv,[syntax:[eventually]]).
```

```
/*==========================================================================
                         Semantic Lexicon Lambda
============================================================================
                         Det Semantics
==========================================================================*/
semLex(det,[type:forall,sem:lbd(g,forall(g))]).
semLex(det,[type:exists,sem:lbd(g, exists(g))]).
semLex(det,[type:def,sem:lbd(g,def(g))]).
/*==========================================================================
                         Verb Semantics
==========================================================================*/
semLex(iv,[symbol:Sym,type:forall,sem:lbd(g,forall(Fla))]):- Fla=..[Sym,g].
semLex(iv,[symbol:Sym,type:exists,sem:lbd(g, exists(Fla) during 'T')]):-
       Fla=..[Sym,g].
semLex(iv,[symbol:Sym,type:init_exists,sem:lbd(g,init_exists(Fla) during 'T')]):-
       Fla=..[Sym,g].

semLex(iv_adj,[symbol:_,type:forall,sem:lbd(p, lbd(g,forall(p@g) ))]).
semLex(iv_adj,[symbol:_,type:exists,sem:lbd(p,lbd(g, exists(p@g) during 'T'))]).
semLex(iv_adj,[symbol:_,type:init_exists,sem:lbd(p,lbd(g, init_exists(p@g) during 'T'))])
    .
semLex(tv,[symbol:Sym,type:forall,sem:lbd(h,lbd(g, forall(Fla)))]):-  Fla=..[Sym,g,h].
semLex(tv,[symbol:Sym,type:exists,sem:lbd(h,lbd(g, exists(Fla) during 'T'))]):-
       Fla=..[Sym,g,h].
semLex(tv,[symbol:Sym,type:init_exists,sem:lbd(h,lbd(g,init_exists(Fla) during 'T'))]):-
       Fla=..[Sym,g,h].

semLex(dtv,[symbol:Sym,type:forall,sem:lbd(h,lbd(g,lbd(f, forall(Fla))))]):-
       Fla=..[Sym,f,h,g].
semLex(dtv,[symbol:Sym,type:exists,sem:lbd(h,lbd(g, lbd(f,exists(Fla) during 'T')))]):-
       Fla=..[Sym,f,h,g].
semLex(dtv,[symbol:Sym,type:init_exists,sem:lbd(h,lbd(g, lbd(f,init_exists(Fla) during 'T
    ')))]):- Fla=..[Sym,f,h,g].
/*==========================================================================
                         Coord Semantics
==========================================================================*/
semLex(ipcoord,[type:conj,sem:lbd(q,lbd(p,(p & q )))]).
semLex(ipcoord,[type:disj,sem:lbd(q,lbd(p,(p v q)))]).

semLex(tncoord,[type:conj,sem:lbd(p,lbd(q,lbd(z,lbd(s,lbd(r,(s@(z@p))@r & (s@(z@q))@r))))
    )]).
semLex(tncoord,[type:disj,sem:lbd(p,lbd(q,lbd(z,lbd(s,lbd(r,(s@(z@p))@r v (s@(z@q))@r))))
    )]).
```

```
semLex(npcoord,[type:conj,sem:lbd(p,lbd(q,lbd(r,(r@p & r@q))))]).
semLex(npcoord,[type:disj,sem:lbd(p,lbd(q,lbd(r,(r@p v r@q))))]).
/*================================================================
                  PN, Temporal Noun, and Number Semantics
================================================================*/
semLex(pn,[symbol:Sym,sem:Sym]).
semLex(tnoun,[symbol:Sym,sem:Sym]).
semLex(number,[symbol:Sym,sem:init_exists(Sym)]).
/*================================================================
                  Adjective Semantics
================================================================*/
semLex(adj,[symbol:Sym,sem:lbd(g,Fla)]):- Fla=..[Sym,g].
semLex(adj_Stable,[sem:lbd(g, exists(g <<-) during 'T')]).
semLex(tadj,[symbol:f,sem:lbd(g,Fla)]):- Fla=..[f,g].
semLex(tadj,[symbol:l,sem:lbd(g,Fla)]):- Fla=..[l,g].
/*================================================================
                  Negation Semantics
================================================================*/
semLex(neg,[sem:lbd(p,lbd(k,~(p@k)))]).

/*================================================================
                  Grammatical Aspect
================================================================*/
grammatical_aspect(-past, perf).
grammatical_aspect(-ed, perf).
grammatical_aspect(-en, perf).
grammatical_aspect(-ing, prog).
grammatical_aspect(none, simple).
/*================================================================
                  Aspectual Classes for some verbs
================================================================*/
aspectual_class(assert,event).
aspectual_class(deassert,event).
aspectual_class(give,event).
aspectual_class(set,event).
aspectual_class(cross,event).
aspectual_class(use,event).
aspectual_class(change,event).
aspectual_class(go,event).
aspectual_class(run,event).
aspectual_class(become,event).
aspectual_class(exceed,event).
aspectual_class(limit,event).
aspectual_class(be,event).
aspectual_class(permit,event).
aspectual_class(occur,event).
aspectual_class(require,event).
aspectual_class(begin,event).
aspectual_class(accept,event).
aspectual_class(allow,event).
aspectual_class(assign,event).
aspectual_class(enable,event).
aspectual_class(exist,event).
aspectual_class(fail,event).
aspectual_class(get,event).
aspectual_class(interleave,event).
aspectual_class(issue,event).
aspectual_class(take,event).
aspectual_class(stall,event).
aspectual_class(produce,event).
aspectual_class(restrict,event).
aspectual_class(sample,event).
aspectual_class(transition,event).
aspectual_class(be,state).
aspectual_class(remain,state).
aspectual_class(stay,state).
aspectual_class(wait,state).
aspectual_class(have,state).
aspectual_class(match,state).
aspectual_class(keep,state).
aspectual_class(hold,state).
/*==========================================================================
                    Selecting a quantifier Type
==========================================================================*/
quantifier_type(simple,state,forall).
quantifier_type(simple,event,exists).
quantifier_type(simple,event,init_exists).
quantifier_type(perf,state,forall).
quantifier_type(perf,event,exists).
quantifier_type(perf,event,init_exists).
quantifier_type(prog,state,forall).
quantifier_type(prog,event,forall).
/*==========================================================================
     Modifing a quantifier when the neg category combines with the vp category
==========================================================================*/
quantifier_modification(exists,forall).
quantifier_modification(init_exists,forall).
quantifier_modification(forall,exists).
quantifier_modification(forall,init_exists).
```