2015

Consistency Checking of Natural Language Temporal Requirements using Answer-Set Programming

Wenbin Li
University of Kentucky, wli9840122@gmail.com

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Wenbin Li, Student

Dr. Jane Huffman Hayes, Major Professor

Dr. Mirosław Truszczyński, Director of Graduate Studies

Consistency Checking of Natural Language Temporal Requirements using Answer-Set Programming

DISSERTATION

A dissertation submitted in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy in the
College of Engineering at the
University of Kentucky

By
Wenbin Li
Lexington, Kentucky

Directors: Dr. Jane Huffman Hayes
Mirosław Truszczyński
Professors of Computer Science
Lexington, Kentucky 2015

ABSTRACT OF DISSERTATION

Consistency Checking of Natural Language Temporal Requirements using Answer-Set Programming

Successful software engineering practice requires high quality requirements. Inconsistency is one of the main requirement issues that may prevent software projects from being success. This is particularly onerous when the requirements concern temporal constraints. Manual checking whether temporal requirements are consistent is tedious and error prone when the number of requirements is large. This dissertation addresses the problem of identifying inconsistencies in temporal requirements expressed as natural language text. The goal of this research is to create an efficient, partially automated, approach for checking temporal consistency of natural language requirements and to minimize analysts' workload.

The key contributions of this dissertation are as follows: (1) Development of a partially automated approach for checking temporal consistency of natural language requirements. (2) Creation of a formal language Temporal Action Language ($TeAL$), which provide a means to represent natural language requirements precisely and unambiguously. (3) Development of a front end to semi-automatically translate natural language requirements into $TeAL$. (4) Development of a translator from $TeAL$ to the ASP language.

Validation results to date show that the front end tool makes the task of translating natural language requirements into $TeAL$ more accurate and efficient, and the translator generates ASP programs that correctly detect the inconsistencies in the requirements.

KEYWORDS: Temporal Requirements, Consistency Checking, Knowledge Representation, Natural Language Processing, Answer Set Program

Author's signature: Wenbin Li

Date: April 30, 2015

Consistency Checking of Natural Language Temporal Requirements using Answer-Set Programming

By
Wenbin Li

Director of Dissertation: Jane Huffman Hayes
Mirosław Truszczyński

Director of Graduate Studies: Mirosław Truszczyński

Date: April 30, 2015

ACKNOWLEDGMENTS

I would like to thank my advisors, Professor Hayes and Professor Truszczyński, for their guidance and advice throughout my doctoral studies. Their support was invaluable throughout my research and career pursuit. I would also like to thank Dr. Marek and Dr. Post for their service on my committee and assistance in my research.

I would like to offer thanks to fellow graduate students Dr. Sultanov, Dr. Hays, Dr. Kidwell, Dr. Kong, Jesse Yanneli and David Brown for their assistance and collaboration.

I would like to thank my parents, Zhengfei Li and Huiqi Shen, for their love and support my entire life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

I proposed, designed, and implemented an efficient and partially automated approach for checking temporal consistency of software requirements given as natural language text. The approach is not fully automated as some tasks require human analyst feedback. My main motivation was to develop methods and techniques to minimize analysts' workload and improve the quality of requirements.

1.1 Motivation

High quality requirements are essential for successful software projects. Studies have shown that 60 - 70% of software project failures are related to defects (or faults) in software requirements [3]. Ambiguity, inconsistency, and incompleteness are common defects that prove harmful to the quality of requirement specifications [10].

A recent study shows that inconsistency accounts for 13% of all requirement defects [45]. Consistency checking must be performed to uncover these defects so that the software system can be implemented as specified. However, consistency checking tasks are generally performed manually, and they are time-consuming and error-prone because of the complexity and the number (dozens or hundreds) of requirements.

Several techniques have been proposed that use formal methods to check consistency automatically [53, 47, 46, 86, 77, 19]. These techniques assume that requirements have already been specified as formalized theories. However, requirements given as formal theories are rare. A vast majority of software projects are specified in natural language. The natural language requirements need to be translated into formal theories. However, manual translation requires a high-level of understanding of the target formalism, something analysts typically do not have. Moreover, the task is labor-intensive and error-prone even if undertaken by analysts with a good preparation in formal methods. Automated approaches

for formalizing requirements also pose many challenges related to the fact that natural language text is often ambiguous, incomplete, and difficult to process. In addition, validating the correctness of translation requires the same high level of familiarity with formal methods as generating a formal theory representing the requirements.

In this thesis I focus on consistency checking of temporal requirements because many software projects implement systems that support real-time operations, and temporal requirements are common in such systems. For example, an e-commerce system requires that a payment be received a specified time prior to submitting an order for processing. Another example is a safety-critical pacemaker system that requires pacing to occur within milliseconds of certain detected events. Moreover, as these examples implicitly suggest, high quality of temporal requirements is essential. Errors in specifying, interpreting, or implementing temporal requirements can lead to disastrous consequences. If two or more requirements related to the pacing of the heart are inconsistent, a negative heart event might not trigger a necessary lifesaving pacing event. One real world example that shows the serious consequences of ignoring the quality of requirements is the AEGIS combat software disaster [32], in which two-hundred and ninety people lost their lives. Some requirements were not specified for the AEGIS software system, and hence were never implemented. Some of the missing AEGIS requirements concerned timing.

In this thesis I propose and describe a semi-automated approach for temporal requirement consistency checking. The key element of this approach is Temporal Action Language ($TeAL$), a formal language that I created. On the one hand, $TeAL$ has syntax aligned with linguistic patterns people use to express temporal constraints in natural language. The syntax facilitates the comprehension and verification of correctness of $TeAL$ statements. On the other hand, the semantics of $TeAL$ support the translation to lower-level logic formalism. The overall approach consists of two main phases: (1) developing a $TeAL$ theory from natural language requirements, and (2) translating the $TeAL$ theory into lower-level logic formalism and using existing tools for processing.

For the first phase, I designed and implemented a tool, $TeALGenerator$, for generating $TeAL$-like statements from natural language requirements. The outputs of $TeALGenerator$ are referred to as "$AlmostTeAL$" because they may contain inaccuracies or may be missing some information. Analysts need to verify $AlmostTeAL$ statements and generate correct $TeAL$ statements from them. For the second phase, I designed and implemented a translator to automatically translate $TeAL$ theory to a lower-level logic formalism, Answer Set Programming (ASP) [66, 72]. I selected an existing tool, $clingcon$ [35], to process the ASP program and determine if there are inconsistencies among the requirements. The $clingcon$ was chosen due to its ability to handle linear constraints on large numeric domains.

1.2 Contributions

The semi-automated approach described in this thesis aims to minimize the time and effort analysts spend on the consistency checking task by reducing the task to that of generating Temporal Action Language ($TeAL$) theories. The contributions include:

1. Temporal Action Language ($TeAL$) as a key component of the approach.
    - $TeAL$ serves as an intermediate level between natural language requirements and low-level logic formalism, bridging the gap between the two abstraction levels.
    - $TeAL$ can be used as a formal language to specify requirements in the first place.
2. $TeALGenerator$, a tool to build $TeAL$-like statements ($AlmostTeAL$) from natural language requirements. $AlmostTeAL$ statements are used as the guideline and analysts can revise these statements into correct $TeAL$ statements.
3. A translator, $TeAL2ASP$, for translating $TeAL$ theories into Answer Set Programming (ASP) [66, 72]. An existing tool, $clingcon$ [35], uses these ASP programs as

input and generates outputs that indicate the consistency (or the lack of consistency) of $TeAL$ theories.

- The correctness of the translation algorithm in $TeAL2ASP$ is proved: analyzing the output of $TeAL2ASP$ indicates if the input $TeAL$ theory is consistent or not.

4. Experimental studies that determine the readability of $AlmostTeAL$ and measure the efficiency of generating $TeAL$ statements with the assistance of $AlmostTeAL$.

1.3 Organization of the Thesis

The remainder of the thesis is organized as follows. Chapter 2 presents the general approach. Chapter 3 discusses related work. Chapter 4 presents the syntax and semantics of the intermediate language $TeAL$. Chapter 5 discusses the front-end tool for generating $TeAL$. Chapter 6 presents the translation from $TeAL$ to the low-level logic formalism. Chapter 7 discusses the empirical studies. Chapter 8 concludes the thesis and outlines future work.

Chapter 2 General Approach

The key idea of the semi-automated approach discussed in this thesis is to use an intermediate language, Temporal Action Language ($TeAL$), to support the process of formalizing requirements given in natural language text.

Natural language requirements are ambiguous and rely on implicit information [16]. It is very difficult, if not impossible, to automatically construct a formal theory based on a set of imprecise requirements and to ensure that the formal theory captures the meaning of these requirements. However, a formal theory that specifies the natural language requirements precisely is necessary for automating the analysis of a software system.

It is an analysts' task to decide if a given formal theory is a precise specification of a set of natural language requirements. However, that requires strong background in logic formalisms. Even if analysts have such background, poor readability of typical logic languages in which formal theories are given makes the validation of the formal theory an ineffective, error-prone and time consuming task.

My approach introduces an intermediate language to bridge the gap between natural language and low-level logic formalism. To address the issues discussed above, the intermediate language must be close in syntax to natural language to support the validation of formal theories given in this language, yet it must have precise semantics to allow automated translation to lower-level logic formalism. To the best of my knowledge, no formal language that satisfies these requirements exists. I introduced Temporal Action Language ($TeAL$), a key component in my approach, as a candidate for the intermediate formalism. The creation of $TeAL$ theories requires analysts' involvement because of the ambiguity of natural language. But the task of checking if theories are consistent can be fully automated due to the formal nature of $TeAL$. I have designed and performed experiments to compare the readability of $TeAL$ and a lower-level logic formalism. The results show that it is easier

for analysts to work with $TeAL$ than with the low-level logic formalism (these experiments are discussed in detail in Chapter 7). I also created $TeALGenerator$, a tool that generates "*close-to-*$TeAL$" ($AlmostTeAL$) statements automatically. Experimental results show that analysts generate $TeAL$ theories more efficiently with the assistance of $AlmostTeAL$ statements. The experiments are discussed in Chapter 7.

Details of the approach are illustrated using the motivating scenario below:
A system consists of three subsystems, A, B, and C. Subsystem C "*controls*" the system and sends data to and receives data from Subsystems A and B. When C is not sending or receiving, it is idle. Subsystem C expects a "*heartbeat*" message from the other two subsystems every N milliseconds. If such a message is not received, Subsystem C begins to perform a sequence of activities to mitigate degradation of the system. Subsystems A and B alternate reading messages from Subsystem C, writing to a hardware device, and responding to messages from Subsystem C. This is done in a timed sequence. Subsystem A must write to the hardware device within K milliseconds of receiving a message from Subsystem C. Subsystem B must write to the hardware device within M milliseconds of receiving a message from Subsystem C.

This scenario describes a software system. A requirement specification document can be created based on this scenario. Some of the requirements this document could contain are listed below:

- R1: The system contains three subsystems A, B, and C.
- R2: Subsystem C can send data to and receive data from Subsystems A and B.
- R3: Subsystems A and B can send heartbeat messages to Subsystem C.
- R4: If a heartbeat message is not received from Subsystem A every N milliseconds, Subsystem C shall initiate degradation actions.

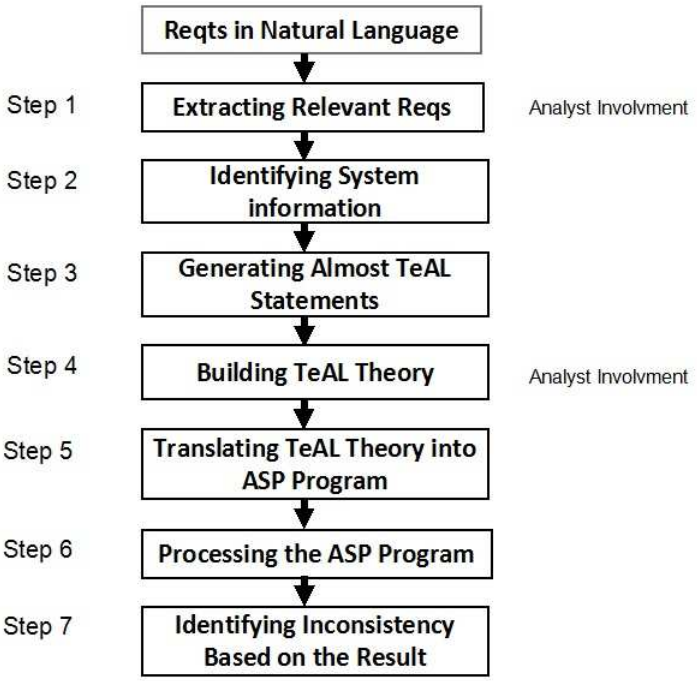


Figure 2.1: Steps in the Consistency Checking Process

- R5: Subsystem A must write to the hardware device within K milliseconds after receiving a message from Subsystem C.

Requirements R1, R2, and R3 do not contain temporal information. Requirements R4 and R5 specify bounds on times when an action may or may not occur, such as "*initiate degradation every N milliseconds*" and "*write within K milliseconds.*" Thus, they are temporal requirements.

As mentioned earlier, $TeAL$ divides the approach introduced in this thesis into two phases: generating $TeAL$ theory from natural language requirements, and processing the $TeAL$ theory to check consistency. The first phase generates $TeAL$ theory based on the requirements above. As Figure 2.1 shows, this phase contains four steps, and only Step 1 and Step 4 require analysts' involvement.

Step 1 uses existing Information Retrieval (IR) techniques [75, 20] to extract requirements that contain temporal information. Most of these requirements contain keywords such as *before* and *within*, or patterns such as *do action every x seconds*. It is feasible to

detect many, if not all, of the temporal requirements based on these keywords and patterns.

Given a set of temporal requirements, non-temporal requirements that are related to them and that might contain relevant system information must be identified as well. An IR tool, REquirements TRacing On target (RETRO) [44], is useful in this task. The RETRO tool can identify requirements that may be relevant to the temporal requirements based on common terms. Analysts must verify whether the requirements found in this step are truly relevant and must ensure that all relevant requirements have been identified. By the end of this step, all requirements that are necessary for modeling the software system are identified.

For Steps 2 and 3, I designed and implemented a tool $TeALGenerator$. Step 2 identifies the system elements that are necessary for generating $TeAL$ theories. Step 3 constructs these elements into $AlmostTeAL$. The front-end tool uses Natural Language Processing (NLP) tools including Stanford Parser [52, 24] and the semantic role labeler Senna [21].

The elements that must be identified in Step 2 for formalizing requirements include *agents*, *actions*, *fluents*, and *temporal constraints*.

An *agent* is a component of a software system that can act and accomplish tasks for the user of the software [76]. In this case, there are three agents: Subsystems A, B, and C.

An *action* is an event that can change the state of a software system. Each action involves an agent that executes it, and has duration and effects. In this case, Subsystems A and B can perform actions such as "*send heartbeat message to Subsystem C,*" "*write to a hardware device,*" and "*receive message from Subsystem C.*" Subsystem C can also perform the actions "*receive heartbeat message*" and "*send message to subsystems A and B.*"

A *fluent* is a property of a software system that can be changed because of the occurrence of actions or the passage of time. For example, "*Subsystem C is idle*" is a fluent, it can be changed by the actions "*receive messages from Subsystem A/B.*"

A *temporal constraint* may specify time bounds on actions, frequencies with which

actions must occur, or the occurrence of actions in time. Such temporal constraints may refer to multiple discrete time points. For example, R4 specifies that the frequency with which the action "*receive heartbeat message from Subsystem A*" should occur is "*once every N milliseconds.*" In R5, the time bound on the action "*write to hardware device*" is "*within K milliseconds after receiving a message from Subsystem C.*"

Step 3 focuses on the relations among the elements identified in Step 2 and uses these relations to construct $AlmostTeAL$ statements. Step 3 defines the preconditions and effects of these elements. For instance, in this step $TeALGenerator$ detects that in R4, if the precondition "*heartbeat message is received every N milliseconds*" is not met, then the consequence is "*subsystem C initiate degradation actions*." (Step 3 is discussed in detail in Chapter 4.)

Step 4 requires the involvement of an analyst whose task is to convert $AlmostTeAL$ statements to $TeAL$ statements that correctly represent input requirements. The involvement includes adding missing elements and removing inaccuracies in the auto-generated $AlmostTeAL$ statements. For instance, at the end of this step analysts will have the following $TeAL$ statement for R5:

$$\begin{aligned} &\textbf{if } received(sysC, msg, sysA) \\ &\quad \textbf{then commence } write(sysA, dev) \\ &\quad \textbf{noLaterThan } K \ millisecond \textbf{ after};\end{aligned}$$

Phase 2 uses a correct $TeAL$ theory as input and outputs a result that indicates whether the requirements are inconsistent. This phase is fully automated and hidden from analysts.

For Step 5, I designed and implemented a translator, $TeAL2ASP$, to translate the $TeAL$ theory into low-level logic formalisms[1]. I chose Answer Set Programming (ASP) [66, 72] as the formalism used in this approach. ASP is a logic-based formalism which maps real-life problems to logic expressions and offers tools to reason with these logic expressions

[1]I acknowledge the assistance of David Brown with the implementation of $TeAL2ASP$.

automatically. For instance, R5 is modeled in ASP by means of several clauses such as:

$$sat(C3,receive(sysA, msg, sysC), write(sysA, dev), C)\text{:-}$$
$$happen(totrue(receive(sysA, msg, sysC)), C1),$$
$$horizon\$ >= time(C) + K, happen(com(write(sysA, dev)), C2).$$

The output of Step 5 is ready to be processed by ASP solvers. The readability of the output is not a concern because analysts will not need to read it. I chose a solver, $clingcon$ [35], as the tool for processing the output of $TeAL2ASP$ because $clingcon$ is designed to handle large numeric domains. The translation to ASP is discussed in detail in Chapter 6.

In Step 6, the output of Step 5 is processed by the ASP solver automatically. The output illustrates if the requirements are consistent.

In Step 7, the result generated by the ASP solver is analyzed and tracked back to the original natural language requirements. The result tells analysts if the requirements are consistent or not. However, analysts may require more details. For example, knowing that there is an inconsistency somewhere among fifty temporal requirements may not be very useful. Identifying five problematic requirements that may be responsible for the inconsistency is much more useful. The task of identifying problematic requirements is included in future work.

Chapter 3 Related Work

This chapter provides an overview of related work. I discuss information retrieval (IR), natural language processing (NLP), requirements specification, model checking, answer set programming (ASP), and requirements validation techniques.

3.1 Information Retrieval and Natural Language Processing

Information retrieval (IR) and natural language processing (NLP) techniques are relevant to the process of generating $AlmostTeAL$. IR techniques are used for extracting requirements that contain necessary information for constructing a $TeAL$ theory and NLP techniques are used for extracting it.

Information Retrieval

IR techniques are useful for searching, analyzing, and extracting domain information from natural language requirements.

The vector space retrieval method is one of the most commonly used information retrieval algorithms [8]. Vector space retrieval can be used to calculate the similarity between the query, a formal statement of information that users seek, and a document, the source of information. The first step in using vector space retrieval is assigning weights to keywords and representing documents as vectors of these weights. Then, the relevancy ranking is calculated based on the frequency of terms (words) from a query in a document. The higher the relevance ranking, the higher the similarity between the query and the document.

Two measures, recall and precision, are used to measure the efficiency of IR algorithms. Recall measures the percentage of the relevant documents that are found, while precision measures the percentage of the found documents that are relevant. Finding documents relevant to the temporal requirements will be efficient if recall and precision are high.

Existing IR techniques can be used in the process of identifying temporal requirements within a set of natural language requirements. Nikora and Balcom [75] developed a program to identify Linear Temporal Logic (LTL) [42] patterns from natural language requirements. The requirements that contain these patterns may share some characteristics, such as temporal prepositions. Nikora and Balcom preprocessed the natural language requirements and applied vector space retrieval. Their experiments showed that LTL patterns were retrieved with high accuracy. Cleland-Huang, Settimi, Zou, and Solc developed another program that can be used to find temporal requirements [20]. Their program identifies non-functional requirements by using special keywords. Since temporal requirements often share similar keywords such as "before" and "after," their program can be tailored to identify temporal requirements.

IR techniques and their application to traceability can be useful in collecting domain information that is related to temporal requirements. Hayes, Dekhtyar, Sundaram, Holbrook, Vadlamudi, and April introduced a tool REquirements TRacing On target (RETRO) [44]. RETRO uses the vector space retrieval method. It is capable of finding the requirements that share keywords with temporal requirements. Analysts can check the requirements found by RETRO and decide if these requirements contain useful information.

RETRO determines if two files are related based on their similarity. Term frequency-inverse document frequency (TF-IDF) is one of the most frequently used weighting methods for vector space models. It is also the weighting method used in RETRO. Term frequency measures how many times a term is used in a document. Inverse document frequency measures how important a term is among a set of documents [8].

Natural Language Processing

Natural language processing (NLP) is another area that is closely related to the process of generating $AlmostTeAL$.

Stanford Parser [52, 24] is one of the most frequently used tools for natural language

processing, it is a Java implementation of a probabilistic natural language parser. A natural language parser is a program that analyzes the grammatical structure of sentences. This research makes use of the following functionalities of Stanford Parser: Parts-of-Speech tagging, chunking, parsing, and the Stanford dependencies. Each are discussed next.

Parts-of-Speech (POS) tagging [84] is the process of marking words or phrases in a text with particular parts of speech. The tagging is based on the context and the definition of the words or phrases. Words are classified as nouns, verbs, adjectives, etc., and phrases as noun/verb phrase, subject/object, etc. POS tagging provides useful information for identifying system elements in this research.

Chunking [52] is another widely used natural language processing technique which separates a sentence into meaningful pieces (e.g., noun groups, verb groups) rather than single words. Chunking does not provide the internal structure of the sentence. My approach makes use of chunking to identify phrases.

Parsing analyzes the grammatical structure of a sentence and generates a parse tree. The parse tree illustrates the syntactic relation among the sentence words. Two types of parse trees are commonly used: constituency-based parse tree and dependency-based parse tree. The constituency-based parse trees contain two kinds of nodes: terminal and non-terminal. All interior nodes are non-terminal nodes (e.g., noun/verb phrase) and all leaf nodes are terminal nodes (e.g., noun/verb). The dependency-based parse trees contain only terminal nodes (e.g., noun/verb). Given a sentence, its dependency-based parse tree contains fewer nodes than its constituency-based parse tree. Parsing and POS tagging are closely related and provide syntactic information in this research.

The Stanford dependencies [52] provide a simple and uniform representation of semantic relations between words. These dependencies are designed to be easily understood. All dependencies are binary and contain the name of the relation, governor, and dependent, where the relation holds between the governor and the dependent. For instance, given a sentence "*The message is sent by the server*" and a dependency $agent(sent, server)$, the

relation is $agent$, the governor is $sent$, and the dependent is $server$. This typed dependency means that $server$ performs the action represented by the passive verb $sent$. Standford dependencies contain more than fifty relations. My approach makes use of Stanford dependencies to understand relations among nouns, verbs, and phrases.

Semantic Role Labeling (SRL) [39] is another natural language processing technique that detects semantic relations. SRL identifies semantic arguments that are related to the verbs (predicates) in a sentence. Moreover, SRL classifies the arguments into different roles. For instance, given a sentence "*server sends a message to the node*," the verb $send$ is identified as the predicate, the role of $server$ is agent (sender), the role of $message$ is theme (what to send), and the role of $node$ is recipient. Senna [21] is a semantic role labeler which also supports POS tagging and chunking. Another semantic role labeler is the Semantic parser [13]. It uses a converter to transform constituent parse trees into dependency graphs. Most actions, fluents, and their arguments are identified by SRL in my approach.

3.2 Requirements Specification

Much research focuses on specifying requirements using formal languages. It is necessary to examine these languages and understand the difference between them and $TeAL$.

Action languages are formal languages that focus on representing actions, their effects, and their preconditions. However, action languages do not support the representation of the temporal constraints in software systems, including the concept of action duration. Baral and Gelfond proposed an action language AL [9]. The semantics of AL are based on the "inertia axiom" assumption. This assumption states that "things remain the same until something happens to make things change." A system specified by AL can be modeled as a transition system. The nodes of the transition system represent the states of the system and the arcs of the transition system represent actions that cause changes. Action languages are fundamental to $TeAL$ because $TeAL$ is constructed as an extension of the action language

AL. I made extension to both the syntax and semantics of AL so that $TeAL$ can describe complex temporal relationships.

Modular Action Language (ALM) [36] is an extension of AL which supports definition of actions and fluents with more detail. The language ALM divides the declaration of actions and fluents into multiple modules. Each action or fluent has its own modules and the information such as preconditions and effects are organized based on these modules.

Giunchiglia and Lifschitzs proposed another action language C [41]. The semantic of this action language is based on the "causality principle." The causality principle claims that "everything must be caused." This principle means that when a proposition is true there must be a reason for it to be true.

Temporal Action Logic (TAL) [27] is a class of logics that supports reasoning about actions and temporal changes. A TAL narrative consists of two parts: the narrative background and the narrative specification. The narrative background contains information that is common to all narratives in a domain, including the generic definition of actions and fluents, and the causal dependencies among the actions/fluents. The narrative specification contains information that is specific to particular instances, such as when actions actually do take place in a narrative, and the initial states. I did not choose TAL as the intermediate language in this thesis because TAL lacks obvious ways to represent complex temporal relationships, such as "*send message within 2 seconds after receiving a message,*" between actions. Another reason is that validating TAL statements is a difficult task for analysts who lack a strong background in logics.

Temporal logics are another class of formalisms that allow users to represent and analyze systems in terms of time. Because of this, temporal logics can be used for formal verification of hardware and software systems [51]. Computational tree logic (CTL) [30] is a temporal logic that uses a tree-like structure as its model of time. This model shows that there are multiple paths in the multiple futures and one of these paths may be realized. Linear temporal logic (LTL) [42] is a temporal logic that uses a structure with a single

future (a path). It is easy to express properties such as "some condition will be true eventually" or "some condition cannot be true until something happens" in LTL. Both CTL and LTL are used by model checking techniques for describing temporal properties. $TeAL$ is not designed based on CTL or LTL because some common temporal requirements (e.g., *perform an action within 10 seconds*) are hard to represent in temporal logics.

Situation calculus [68] and event calculus [55] are also formalisms that support temporal reasoning. The basic elements of situation calculus are actions, fluents, and situations. A situation represents a sequence of actions. The symbol do represents the situation resulting from an action. For instance,

$$do(send(message), do(generate(message), S_0))$$

represents the situation resulting from "*generating and sending a message*." Additionally, like the other formalisms that reason about actions and changes, situation calculus uses formulas to represent action effects and preconditions. Event calculus allows users to specify temporal information such as: the value of fluents at given time points, and the actions that take place at given time points. The predicate $HoldsAt(f, t)$ is used to determine if the value of a fluent f is true at time point *t*. The predicates $Initiates(a_1, f, t_1)$ and $Terminates(a_2, f, t_2)$ represent that the action a_1 makes f true at t_1 and a_2 makes f false at t_2. Event calculus also uses $Happens(a, t)$ to represent that *a* happens at *t*, without describing its effect. The problem with using situation calculus and event calculus is similar to TAL: these languages are not easy to read and validate. However, the predicates discussed above are useful in describing temporal information in ASP programs. In the approach introduced in this thesis, $TeAL$ theories must be translated into ASP programs. These ASP programs contain predicates that are close to the predicates discussed above.

Software Cost Reduction (SCR) [48] is a well known requirement specification technique that describes the system as a state transition machine. An SCR requirement specification can be used to specify various requirements, such as functional and temporal require-

ments. However, the distance between natural language requirements and a state transition machine prevents SCR from being used as the intermediate language in this thesis.

There are languages designed for formalizing policies. Some policies specify when certain events may or may not occur. These policies are very close to temporal requirements. Gelfond and Lobo proposed a language for specifying policies [38]. This language supports the representation of fluents, actions, and two types of policy statements: Authorization Policy, which describes whether certain actions are permitted; and Obligation Policy, which describes whether certain policies are obligated. Another logic-based policy specification language is introduced by Craven, Lobo, Ma, Russo, Lupu, and Bandara [22]. The major difference from the previous study proposed by Gelfond and Lobo is that this work allowed policies to include temporal information.

3.3 Model Checking and Answer Set Programming

Model checking and answer set programming (ASP) techniques are closely related to the processing of the low-level logic formalism in my approach. I selected ASP as the target formalism to which $TeAL$ theories are translated. I also considered translating $TeAL$ theory into a program accepted by a model checker.

Model Checking

Model checking is one of the most commonly used techniques for validating temporal constraints [69]. Model checking techniques are more frequently used in analyzing hardware systems, but can also be used to verify software systems. The techniques can be used to determine whether properties hold based on the specification of a system.

Clarke, Emerson, and Sistla's study [19] is a pioneering work in the model checking field which provided an early example of using temporal logic in model checking. Clarke, Emerson, and Sistla modeled a concurrent system as a finite-state machine and created the specifications of this system in temporal logic. They then checked if any property viola-

tion can occur in the finite-state machine. The size of the finite-state machine was large: the finite-state machine had hundreds of states. The results showed that model checking can be applied to such large systems with reasonable efficiency. This study gave rise to much subsequent research. Bounded model checking [12] is one of them. This technique addresses the state explosion problem. The state explosion problem limits the applicability of model checking because whenever the complexity of a system increases, the size of the system model increases exponentially. This problem must be addressed, otherwise the efficiency of applying model checking to most real-world problems will be unacceptable. Unlike the study discussed above, the bounded model checking algorithms check whether a property is violated within a fixed number of steps instead of assuming that the system may run forever. This technique combines model checking and satisfiability solving techniques because it specifies the system as a boolean satisfiability problem. The bounded model checking algorithm determines that, for every system model and every property, there exists a completeness threshold k such that if there is no violation detected within k steps, then there is no violation for all boundaries greater than the threshold. Clarke, Biere, Raim and Zhu studied the complexity of computing this threshold [18]. The approach introduced in this thesis is close to the concept of bounded model checking, because the approach also checks if there is inconsistency within a threshold (it is called time horizon in this thesis) instead of checking consistency for arbitrarily long time. Details of consistency checking are discussed in Chapter 4.

Model checking techniques can also model the system as timed automata. Yovine examined different model checking techniques that specified real-time systems as timed automata [89]. A timed automaton uses a set of clocks to represent the temporal properties in real-time systems. The complexity of model checking in Yovine's study increases exponentially with the number of clocks used in the timed automaton.

Many model checkers have been implemented. The $NuSMV$ model checker [17] is a model checking tool based on Binary Decision Diagrams. It allows users to represent prop-

erties to be verified in LTL or CTL. It also supports LTL extended with "*past operations*" (e.g., $\mathbf{Y}$ for *previous* and $\mathbf{X}$ for *next*). The $SPIN$ model checker [50, 49] describes the system to be verified in Process Meta Language, and the properties to be verified in LTL. $SPIN$ can also simulate possible execution paths through the system. The $Uppaal$ toolbox [56] is a model checker designed for verification of real-time systems. Unlike the other model checkers discussed in this section, $Uppaal$ models the system in timed automata.

Answer Set Programming

Answer-set programming (ASP) [66, 72] is a computational knowledge representation formalism that is used to represent and solve search problems. ASP is based on the stable model (answer set) semantics of logic programming [37], and search problems are reduced to computing answer sets.

An answer set program consists of rules of the form:

$$A \leftarrow B_1, \ldots, B_m, not\ C_1, \ldots, not\ C_n. \tag{3.1}$$

where $A, B_1, \ldots, B_m, C_1, \ldots, C_n$ are grounded atoms. The left side of the arrow is the *head*, and the right side is the *body*. If the body is empty, then the rule is a *fact*. If the head is empty, then the rule is a *constraint*. A stable model of an answer set program has the following property: for every rule in the answer set program, if all of the positive atoms in the body ($B_1, \ldots, B_m$) are in the stable model, and none of the negative atoms in the body ($C_1, \ldots, C_n$) is in that stable model, then A must be included in the stable model.

Answer set programs also include *choice* rules such as:

$$\{b, c, d\} \leftarrow a. \tag{3.2}$$

This rule means: if *a* is in a stable model, then one of *b,c,d* must be included in that stable model as well. The rule above can be extended with numerical bounds:

$$1\{b, c, d\}2 \leftarrow a. \tag{3.3}$$

This rule means: if *a* is in a stable model, then at least one and at most two of the atoms *b,c,d* must be included in that stable model. The definition of a stable model was extended to the rules above by Niemelä [74].

Answer sets are generated by answer set solvers, the software that processes answer-set programs. The answer set solvers are built based on the technology of SAT solvers [62, 71]. Many answer set solvers have been implemented, including: $smodels$ [73], $assat$ [64], $clasp$ [34], and $aspps$ [29].

Although answer set solvers are capable of solving search problems, these solvers are not sufficiently efficient when the input program has many variables with large domains.

A solver grounds a program with variables before processing it. Grounding means replacing a variable with all of its possible values. This may cause a combinatorial explosion of clauses. For instance, given a rule:

$$p(X, Y) \leftarrow q(X),\ r(Y). \tag{3.4}$$

where both *X* and *Y* have one hundred possible values, grounding removes these two variables by replacing this rule with ten thousand rules. Variables like the ones in the example generate too many ground instantiations and the answer set solver's capability may be exceeded. Optimization algorithms have been introduced for grounding techniques. Mellarkod, Gelfond, and Zhang introduced an approach that integrates ASP and constraint logic programming [70]. This approach presented a knowledge representation language $AC(C)$, an extension of ASP, and a new solver designed for $AC(C)$. The solver, $ACSolver$, created constraint programs using variables with large ranges and solved these constraint programs using an integrated constraint solver. Another variant of answer set solver is $clingcon$ [35]. The $clingcon$ solver also combines answer set programming with constraint solving and prevents the solver from being overwhelmed by a huge number of grounding instantiations.

3.4 Requirements Validation

Requirements validation techniques attempt to prevent or identify inconsistency, incompleteness, and incorrectness in a set of requirements. Much research uses formal methods to automate the process of analyzing software systems. These techniques should be examined because the approach introduced in this thesis uses formal methods as well.

Many requirements validation techniques use model checking to validate software systems. One study used the $SPIN$ model checker to automatically verify UML Statechart Diagrams [57]. This study focused on the translation from UML Statechart Diagrams to $PROMELA$, the input language of $SPIN$. The study covered several important properties for real-time systems, such as concurrent behavior, sequentialization, and non-determinism. Knapp, Merz, and Rauh [54] proposed a prototype tool, $HUGO/RT$, for performing the model checking task. This study focused on checking real-time object-oriented system design. Instead of modeling the system as a finite-state machine, Knapp *et al.* [54] converted timed state machines and temporal constraints into observer timed automaton and timed automata that represent the model, respectively. The timed automata were checked against the observer timed automaton. The major difference between this thesis and the studies above is that this thesis assumes that requirements are given in natural language, while these studies assumed that requirement formal specifications have already been created.

There are research projects that use ASP to validate software systems. Lee and Palla proposed an approach for reformulating situation calculus and event calculus in an ASP program [58]. Lee and Palla proved that reasoning in situation calculus and event calculus can be reduced to computing answer sets using existing answer set solvers. Lee and Palla also developed a program to reformulate a TAL narrative in an ASP program [59]. They introduced the translations from TAL to both $clasp$ and $clingcon$. The results illustrated that $clingcon$ processes the translated TAL program more efficiently because $clingcon$ has advantages in handling large numeric domains. The approach introduced in this thesis uses $TeAL$ instead of TAL, but the $TeAL$ theories represent "*time*" as large numeric do-

mains, and $clingcon$ handles the "*time*" domain better than $clasp$. Baral *et al.* discussed how to represent actions, their effects, and their precondition in ASP [9]. The study used predicates $occur(\ldots)$ to represent that an action happens at a certain time, and predicates $holds(\ldots)$ to represent that a system property holds at a certain time. In this thesis, $TeAL$ theory reuses Baral and Gelfonds' Action Language [9] to describe the effects and preconditions of actions. This information is translated into ASP in a similar way.

Some requirements validation techniques do not rely on ASP or model checking. Giordano, Martelli, and Schwind [40] proposed an approach to validate planning problems. The study modeled planning problems in Dynamic Linear Time Temporal Logic ($DLTTL$), an extension of temporal logic, and represented the $DLTTL$ theories as satisfiability problems. Heitmeyer, Labaw and Kiskis [47] developed a program for consistency checking. They assumed that requirements are specified in SCR, and they generated finite-state automaton based on these SCR requirements. This technique can automatically detect errors including type errors, non-determinism, and circular definitions. Duran, Ruiz, Bernárdez, and Toro proposed a technique that uses XML and EXtensible Stylesheet Language Transformation ($XSLT$) to perform automatic verification [28]. They developed a model of a software system in XML and used $XSLT$ as the requirements verification language. Heimdahl and Leveson developed a program to check completeness and consistency of state-based requirements [46]. In this program, the requirements are formalized in Requirement State Machine Language ($RSML$). $RSML$ is very similar to Statechart, a diagram that extends state-transition diagrams [43].

Li, Viriyasitavat, Ruchikachorn, and Martin [23] presented an automated framework for verifying service workflow requirements using propositional logic. The requirements are supposed to be given in $SWSpec$ [87], the requirements specification language in service workflow environments. The $SWSpec$ describes the sequence of events using Computational Tree Logic (CTL) [30]. The framework automates the process of verifying $SWSpec$ formulas by translating them into propositional logic. The concept of modeling require-

ments using an intermediate formal specification and translating the specification into lower level logic formalism is the same as my approach.

Mashkoor and Matoussi [67] proposed an approach to validate a semi-formal requirement model. The semi-formal requirement model is difficult for customers to verify without any assistance. This approach tries to solve the problem through a gradual introduction of formalism into the requirement model. First, this approach uses $KAOS$ (Knowledge Acquisition in autOmated Specification) [85] to analyze the requirements and build a data model based on first-order temporal logic. Then the $KAOS$ model is translated into an Event-B [6] formal specification, which is based on first order logic. Their approach then validates the Event-B specification automatically. The idea of gradual introduction of formalism is similar to the approach introduced in this thesis.

Like the ASP and model checking techniques discussed above, the major difference between those studies and this thesis is that analysts' workload is not considered. Correct formal specifications are assumed to be inputs, but generating and validating these specifications is a tedious task. In fact, improving the efficiency of generating $TeAL$ theories is one of the key problems addressed by this thesis.

Cabral and Sampaio [14] introduced an approach for formalizing the use cases given in a controlled natural language Attempto Controlled English (ACE) [33]. Cabral and Sampaio assumed that the use cases were given in ACE. The advantage of using controlled natural languages is that these languages are close to natural language, have a formal syntax and semantics, and can be translated into an existing formal language. Cabral and Sampaio's approach translates the use cases into a formal model, which can be used as input for a model checker. Processable English ($PENG$) [80] is another controlled natural language that is close to ACE. Schwitter, Ljungberg, and Hood [81] developed a text editor that assists users in creating $PENG$ sentences. The idea of using controlled natural language as the intermediate level between natural language and low-level logic formalism is very close to the use of $TeAL$ in my approach. However, my approach also concerns the

process of creating $TeAL$ statements, including identifying system information from natural language requirements and constructing $TeAL$ using such information. It is possible to use controlled language requirements instead of natural language requirements as the input to my approach, but this is not required.

There are research projects that use Natural Language Processing techniques to validate software requirements given in natural language. Fliedl, Kop, Mayr, Winkler, Weber, and Salbrechter [31] introduced an approach for the linguistic analysis of requirements texts. This approach uses semantic tagging and chunk-parsing techniques to identify system information from natural language text. The information this approach identifies includes *actor* and *condition*, which is close to the *agent* and *precondition* I want to identify in my approach.

Deeptimahanti and Babar [26] described a tool, UML Model Generator from Analysis of Requirements ($UMGAR$), for generating UML models from natural language requirements. Deeptimahanti and Babar extract the necessary information, *actors* and their *actions*, using natural language processing techniques such as Stanford parser [52, 24]. The same type of information is also identified in my approach.

Weston, Chitchyan, and Rashid [88] proposed a tool framework for automatically processing natural language requirements into a formalized model. This tool framework includes a natural language processing tool $ARBORCRAFT$ [7] and $EAMiner$ [78], a tool for identifying the parts of a program that affect other parts of the system in natural language documents. $ARBORCRAFT$ measures the similarity among requirement documents, and creates a feature tree based on these similarities. Analysts have the option to modify the feature tree based on their domain knowledge. My approach identifies the related non-temporal requirements by measuring their similarity among temporal requirements. My approach requires analysts to provide feedback to the results as well. $EAMiner$ uses grammatical patterns in this process. For example, "*mobile phones*" can be detected by the pattern "*noun with qualifiers*," where qualifiers can be adjectives (e.g., mobile). In

my approach the phrases with specific patterns (e.g. *within* followed by a time period) are identified as well.

Chapter 4 Temporal Action Language $TeAL$

As mentioned earlier, Temporal Action Language $TeAL$ is a critical component in the approach discussed in this thesis. The $TeAL$, as an intermediate language that bridges the gap between natural language and logic formalism, must satisfy the following requirements: (1) have a syntax close to natural language expressions appearing in requirement statements, and (2) have a precise semantics.

I designed $TeAL$ as an extension of the action language AL [9]. I chose AL as the basis of $TeAL$ because detecting inconsistencies means finding the logical or temporal conflicts among actions, and the action language AL is designed to describe the effects of the actions and to reason about them. Another advantage of AL is its intuitive syntax. Consequently, analysts do not need much training to interpret or write $TeAL$ statements. The limitation of AL is that AL is created for describing the effects of the actions and not for representing temporal constraints. I extended the syntax of AL to support the temporal constraints in temporal requirements so that $TeAL$ can meet the requirement on expressiveness.

4.1 Syntax of $TeAL$

A $TeAL$ theory is a triple $\Delta = \langle \Sigma, AD, TC \rangle$, where Σ is the *signature* of the theory, AD is an *action description* that defines actions and their effects, and TC is the set of *temporal constraints*. The pair $\langle \Sigma, AD \rangle$ can be viewed as a theory in AL [9] with modifications that will be discussed below. The AL theory $\langle \Sigma, AD \rangle$ does not contain any temporal information.

Signature

Structured similarly to an AL signature, a $TeAL$ signature $\Sigma = \langle S, C, FL, AC, AG \rangle$ defines the names of domains (the set S of sorts), the entities of each domain (the set C of

constants), the names of system properties (the set FL of fluents), and the names of actions (the set AC of actions). One difference between $TeAL$ and AL is that Σ also identifies sorts whose entities can perform actions (the set AG of agents). I use a series of statements to describe the signature of a $TeAL$ theory. Each statement involves a keyword.

A *sort declaration* has the form:

$$\textbf{sort } s_1, \ldots, s_k; \tag{4.1}$$

where $s_1, \ldots, s_k$ are names of sorts. For example, the statement:

$$\textbf{sort } server,\ node; \tag{4.2}$$

specifies two sorts: server and node. These two sorts can also be declared in two separate statements.

Constant declarations define domains of sorts. A constant declaration has the form:

$$\textbf{constant } s\ con_1, \ldots, con_k; \tag{4.3}$$

where s is a sort declared in a sort declaration and $con_1, \ldots, con_k$ is a list of constants. For example, the statement:

$$\textbf{constant } node\ nodeA,\ nodeB,\ nodeC; \tag{4.4}$$

declares that the $node$ sort consists of three elements: $nodeA$, $nodeB$, and $nodeC$. All constants in $TeAL$ are defined in this way.

Agents are special sorts in that their elements can perform actions. Thus, AG is a subset of S. An *agent declaration* has the form:

$$\textbf{agent } ag_1, \ldots, ag_k; \tag{4.5}$$

where $ag_1, \ldots, ag_k$ are sorts that have been declared in the sort declaration statements. For example, the statement:

$$\textbf{agent } server,\ node; \tag{4.6}$$

specifies that the elements of the sorts $server$ and $node$ are agents and so, can perform actions.

Fluents are used to describe logical properties of a system at any given time. Each of them can be assigned a logical value $true$ or $false$. Fluents can be *inertial* or not. If a fluent is inertial, then its logical value does not change unless an action that affects it is performed. If a fluent is not inertial, then its logical value can change without the occurrence of any action. For example, non-inertial fluents may change due to passing time. A *fluent declaration* has the form:

$$\textbf{fluent}\ fluentName(s_1, \ldots, s_k); \tag{4.7}$$

where $fluentName$ is a string that represents a fluent name and $(s_1, \ldots, s_k)$ is a sequence of sort names defining the arguments of the fluent. A fluent may or may not have attributes. The sequence above is empty when the fluent has no attributes. For example, the statement:

$$\textbf{fluent}\ connected(server, node); \tag{4.8}$$

declares a fluent with two arguments of sorts $server$ and $node$. The attributes can be instantiated as specific constants based on the constant declarations.

Actions are specified in the same way as fluents except that the first attribute of an action must be its agent. An *action declaration* has the form:

$$\textbf{action}\ actionName(s_1, \ldots, s_k); \tag{4.9}$$

where $actionName$ is a string that represents an action name and $s_1, \ldots, s_k$ is a list of sort names, in which s_1 must be an agent sort. For example:

$$\textbf{action}\ (server, node); \tag{4.10}$$

declares an action to *establish a connection*. This action is executed by a server.

Action Description

The action description (AD) of a $TeAL$ theory uses statements introduced in AL [9] to specify the *action domain*: the causal dependencies among actions and the relationships among fluents, the prerequisites of actions, and action effects. The three types of statements included in AD are:

State constraints	L **if** P	(4.11)
Dynamic causal laws	a **causes** E **if** P	(4.12)
Executability conditions	**impossible** $a_1, \ldots, a_k$ **if** P	(4.13)

where E, L, P are lists of fluents and their negations, and a, $a_1, \ldots, a_k$ are actions. Expression (4.11) captures the constraint that every state satisfying P must also satisfy L. Expression (4.12) specifies the effects E of action a executed in a state satisfying P. Finally, expression (4.13) specifies that the conditions P must not hold in a state in order for actions $a_1, \ldots, a_k$ to be executable.

Below, I show the AD part of the $TeAL$ theory that represents a system which contains three subsystems A, B, and C. Subsystem C can send messages to Subsystems A and B, A and B can also send messages to C.

% A message can not be $received$ if it is not $sent$.

impossible $receive(sysC, msg, sysA)$ **if not** $sent(sysA, msg, sysC)$;
impossible $receive(sysC, msg, sysB)$ **if not** $sent(sysB, msg, sysC)$;
impossible $receive(sysA, msg, sysC)$ **if not** $sent(sysC, msg, sysA)$;
impossible $receive(sysB, msg, sysC)$ **if not** $sent(sysC, msg, sysB)$;

% The $sent$ fluent is set to $true$ as an effect of the $send$ action.

$send(sysA, msg, sysC)$ **causes** $sent(sysA, msg, sysC)$;
$send(sysB, msg, sysC)$ **causes** $sent(sysB, msg, sysC)$;

$send(sysC, msg, sysA)$ **causes** $sent(sysC, msg, sysA)$;

$send(sysC, msg, sysB)$ **causes** $sent(sysC, msg, sysB)$;

% The $sent$ fluent is set to $false$ as an effect of the $receive$ action.

$receive(sysA, msg, sysC)$ **causes not** $sent(sysC, msg, sysA)$;

$receive(sysB, msg, sysC)$ **causes not** $sent(sysC, msg, sysB)$;

$receive(sysC, msg, sysA)$ **causes not** $sent(sysA, msg, sysC)$;

$receive(sysC, msg, sysB)$ **causes not** $sent(sysB, msg, sysC)$;

% If a *sent* message has not been received, it cannot be sent again.

impossible $send(sysC, msg, sysA)$ **if** $sent(sysC, msg, sysA)$;

impossible $send(sysC, msg, sysB)$ **if** $sent(sysC, msg, sysB)$;

impossible $send(sysA, msg, sysC)$ **if** $sent(sysA, msg, sysC)$;

impossible $send(sysB, msg, sysC)$ **if** $sent(sysB, msg, sysC)$;

Initial Constraints. To define the initial conditions of a system in AD, I use *initial constraints*. *Initial constraints* are not a part of AL, where the initial conditions of a system are specified differently.

An *initial constraint* is a statement:

$$\textbf{initially } L; \tag{4.14}$$

where L is a fluent or its negation. By default the value of a fluent is considered to be false unless an initial constraint specifies it to be true. For example:

$$\textbf{initially } inMode(system, idle);$$

declares that the system is in the "*idle*" mode when the system starts.

Temporal Constraints

The third component TC in a $TeAL$ theory is the key feature that distinguishes $TeAL$ from AL. Expressions of TC specify temporal constraints and action durations.

Prompts. A $TeAL$ theory is capable of describing actions in more detail than AL. While AL disregards the amount of time that an action takes, $TeAL$ allows for specifications of the durations of actions, and the moments when actions start or end. $TeAL$ introduces *prompts* to indicate the time moments when the corresponding events occur. The basic form of a prompt is:

$$prompt\_operator\ action \tag{4.15}$$

where the $prompt\_operator$ can take the values **commence** and **terminate**. The prompt operator **commence** represents the time moments when actions start. The prompt operator **terminate** is used to indicate the time moments when actions are completed. For example, the prompts:

$$\mathbf{commence}\ send(serverA, msg, nodeA)$$
$$\mathbf{terminate}\ send(serverA, msg, nodeA)$$

indicate the time moments when the action $send(serverA, msg, nodeA)$ was initiated and completed, respectively.

It should be noted that I assume that in $TeAL$ theories time is discrete, and I represent time moments as consecutive integers. The distance between two neighboring time moments represents the smallest unit of time appearing in the $TeAL$ theory. At present I assume that only one time unit is used in all temporal constraints. The support of multiple time units is part of the future work.

Consecutive Prompts. It is possible that temporal relationships involve the same action that is performed multiple times. The keywords **next** and **previous** are used in $TeAL$ to

refer to the closest occurrence of the action after and before the time moment when the theory is being examined. I call this time moment the *current time moment*. This time moment is discussed in detail later when I describe the semantics of $TeAL$. The keywords **next** and **previous** are also useful when there is a need to mention two successive occurrences of the prompt in the same $TeAL$ statement. The general form of a consecutive prompt is:

$$prompt\_operator\ \mathbf{next} \mid \mathbf{previous}\ Action \tag{4.16}$$

For example, the expression

$$\mathbf{terminate\ previous}\ (serverA, nodeA)$$

defines a prompt that indicates the most recent occurrence of $\mathbf{terminate}\ (\ldots)$ strictly before the *current time moment*. Similarly,

$$\mathbf{commence\ next}\ (serverA, nodeA)$$

indicates the earliest occurrence of $\mathbf{commence}\ (\ldots)$ strictly after the *current time moment*. It should be noted that the current syntax does not allow the iteration of **next** or **previous**. The introduction of iterated **next** and **previous** operators is part of future work.

Temporal Conditions. The basic component of a temporal constraint is a *temporal condition*. Temporal conditions are used to represent temporal relationships between the occurrence times of two events. The temporal conditions use keywords such as **at**, **after**, **noLaterThan**, and **startTime** to specify the relationships and the times. For example, the temporal condition

$$\begin{aligned}&\mathbf{terminate\ next}\ send(sysA, msg, sysC)\\ &\quad \mathbf{at}\ X\ second\ \mathbf{after}\\ &\quad \mathbf{terminate}\ send(sysA, msg, sysC)\end{aligned}$$

represents the requirement that "*sysA sends a message to sysC once every X seconds*." Similarly, the temporal condition

$$\textbf{commence } send(sysA, msg, sysC)$$
$$\textbf{noLaterThan } X \ second \textbf{ after startTime}$$

represents the requirement that "*sysA sends a message to sysC within X seconds after the system starts.*" The Backus Normal Form for a temporal condition follows:

$\langle tempCond \rangle ::= \langle timeRef_1 \rangle$ [$when_1$] | [$when_2$]

$when_1 ::= \langle timeComp \rangle \langle timeMod \rangle$

$when_2 ::= \langle timeComp \rangle [\langle timeMod \rangle] \langle timeRef_2 \rangle$

$\langle timeComp \rangle ::= \textbf{earlierThan} \mid \textbf{at} \mid \textbf{laterThan} \mid \textbf{noEarlierThan}$

$\mid \textbf{noLaterThan}$

$\langle timeMod \rangle ::= X \ unit \ \textbf{before} \mid X \ unit \ \textbf{after}$

$\langle timeRef_1 \rangle ::= \langle prt \rangle \mid \langle L \rangle$

$\langle timeRef_2 \rangle ::= \langle prt \rangle \mid \textbf{startTime} \mid L$

$\langle unit \rangle ::= second$

$\langle L \rangle ::=$ fluent or its negation

Here X is a positive integer. I use $timeRef_1$ and $timeRef_2$ to specify time moments. The term $\textbf{startTime}$ represents the time moment when the system starts; prt determines the time moment when the corresponding event occurs; a fluent F appearing in a temporal condition represents the time when a change that makes F to be $true$ occurs. Similarly, the negation of a fluent in a temporal condition represents the time when a change that makes F to be $false$ occurs [61]. The time given by $timeRef_2$ can be modified by $X \ unit$ $\textbf{before}$ or $X \ unit$ $\textbf{after}$ ($timeMod$), where Term $unit$ refers to the time unit that is used in the $TeAL$ theory. As I have discussed above, for now I assume that a $TeAL$ theory only uses one time unit. In the examples below, that unit is $second$.

I use the keywords **earlierThan**, **noLaterThan**, **laterThan**, **noEarlierThan** and **at** ($timeComp$) to specify the relationship between the time moments determined by $timeRef_1$ and $timeMod\ timeRef_2$. For example, in

$$\textbf{terminate}\ dropConn(serA, nodeB)$$
$$\textbf{at}\ 5\ second\ \textbf{after commence}\ (serA, nodeA)$$

the temporal condition specifies the relation between the time moments determined by the prompt **terminate** $dropConn(serA, nodeB)$ ($timeRef_1$) and the prompt **commence** $(serA, nodeA)$ ($timeRef_2$). The relation is **at** $5\ second$ **after** ($timeComp$ and $timeMod$). Here I only provide intuitive explanation of this temporal condition. The detailed semantics is discussed in next section. This temporal condition means that whenever $serA$ starts to establish connection to $nodeA$ (**commence** $(serA, nodeA)$), $serA$ will drop the connection to $nodeB$ (**terminate** $dropConn(serA, nodeB)$) at 5 seconds later. Another temporal condition

$$\textbf{terminate next}\ dropConn(serA, nodeB)$$
$$\textbf{at}\ 5\ second\ \textbf{after commence}\ (serA, nodeA)$$

means that whenever the prompt **commence** $(serA, nodeA)$ occurs, the next occurrence of **terminate** $dropConn(serA, nodeB)$ must be at 5 seconds later. If the temporal condition is

$$\textbf{terminate next}\ dropConn(serA, nodeB)$$
$$\textbf{at}\ 5\ second\ \textbf{after}$$

then this temporal condition is interpreted with respect to an implicit time moment representing the "*current time moment*." The time moment determined by **terminate next** $dropConn(serA, nodeB)$, $timeRef_1$, is the first time after the "*current time moment*" when **terminate** $dropConn(serA, nodeB)$ occurs. The temporal condition above is useless by

itself and may not be used without context, because the "*current time moment*" is not specified. However, in a temporal constraint that consists of multiple temporal conditions, the "*current time moment*" may be specified by other temporal conditions. For example, $TeAL$ expresses the temporal requirement "*A connected node should re-identify itself to the server within 5 seconds after the connection is established, or the server will drop the connection within 10 seconds*" as

if terminate $connect(serA, nodeA)$ **and**
not terminate next $(nodeA, serA)$ **noLaterThan** 5 $second$ **after**
then terminate $dropConn(serA, nodeA)$
noLaterThan 10 $second$ **after**;

In this example, the "*current time moment*" of the temporal condition

not terminate next $(nodeA, serA)$ **noLaterThan** 5 $second$ **after**

is determined by the prompt **terminate** $connect(serA, nodeA)$. This prompt also determines the "*current time moment*" of the temporal condition

terminate $dropConn(serA, nodeA)$ **noLaterThan** 10 $second$ **after**

Temporal Constraints. A *temporal constraint* is an expression:

$$\textbf{if } A_1 \textbf{ and } \ldots \textbf{ and } A_k \textbf{ then } B_1 \textbf{ or } \ldots \textbf{ or } B_m; \tag{4.17}$$

where A_1, ..., A_k and B_1, ..., B_m are *temporal conditions*, their negations, or $true$ and $false$. Based on propositional logic, temporal constraints that contain other boolean combinations of temporal conditions can be represented as a collection of the temporal constraints above. Thus, $TeAL$ does not provide explicit ways to model them directly.

Below, I show the TC part of the $TeAL$ theory that represents the example requirements from Chapter 2. The requirement R4, *If a heartbeat message is not received from Subsystem*

A every K seconds, Subsystem C shall initiate degradation actions, can be written as:

$$\textbf{if terminate previous } send(sysA, msg, nodeC)$$
$$\textbf{at } K \ second \textbf{ before and not } received(sysC, msg, sysA)$$
$$\textbf{then commence } degrade(sysC);$$

Similarly, the requirement R5, *Subsystem A must write to the hardware device within K seconds after receiving a message from Subsystem C*, can be written as a temporal constraint that only consists of one temporal condition:

$$\textbf{commence next } write(sysA, dev)$$
$$\textbf{noLaterThan } K \ second \textbf{ after terminate } receive(sysC, msg, nodeA);$$

Duration Specification

$TeAL$ uses *duration specifications* to specify the duration of actions:

$$\textbf{duration } Action \ x \ unit; \tag{4.18}$$

where x is a positive number and $unit$ refers to time units such as $millisecond$ or $second$.

Common Shorthands

I allow the following shorthands in $TeAL$:

$$\textbf{impossible } pr_1, \ldots, pr_k \textbf{ if } C; \tag{4.19}$$

where C is a temporal condition or its negation. This is equivalent to:

$$\textbf{if } pr_1 \textbf{ and } \ldots \textbf{ and } pr_k \textbf{ and } C \textbf{ then } false; \tag{4.20}$$

I also allow to write:

$$pr \textbf{ causes } C_2 \textbf{ if } C_1; \tag{4.21}$$

where C_1 and C_2 are temporal conditions or their negation. This is equivalent to:

$$\textbf{if } pr \textbf{ and } C_1 \textbf{ then } C_2; \tag{4.22}$$

4.2 Semantics of $TeAL$

As mentioned above, $TeAL$ contains three parts: the *signature* (Σ), the *action description* (AD), and the set of *temporal constraints* (TC). The first two parts, Σ and AD, do not involve temporal information. They are concerned with the states of the system and the effects of actions. The semantics of $\langle \Sigma, AD \rangle$ is based on the semantics of AL [9]. It uses the concept of a transition graph of an AL theory. The transition graph T shows all possible ways for the system described by the $TeAL$ theory Δ to evolve.

Because $TeAL$ supports different action durations, it is necessary to specify when actions start or end, and the temporal relationship between these two events. Since both the start and end of an action change the system, these two events can be viewed as two actions. These new actions have their own preconditions and effects, such as "*once an action is started, it is in progress,*" and "*an action can be finished only if it has already started*." A complete $TeAL$ theory must specify these preconditions and effects. As mentioned earlier, TC uses prompts, $\mathbf{commence}\ Action$ and $\mathbf{terminate}\ Action$, to specify the times when $Action$ starts and ends. These prompts can be used to represent the new "*start*" and "*end*" actions. The use of prompts as new actions gives us a connection between the occurrence of actions and the times when these actions occur.

To formalize the connection, I introduce the *normalized* $TeAL$ theory

$$\Delta^N \ = \ \langle \Sigma^N, AD^N, TC \rangle$$

based on the original $TeAL$ theory $\Delta = \langle \Sigma, AD, TC \rangle$. The normalized signature Σ^N replaces the original actions in Σ with the new ones (prompts), and defines additional fluents that are associated with the prompts. The normalized action description AD^N differs from AD. It extents AD by new statements that specify the relationships among the prompts. The temporal constraints do not change. I will now give a precise description of Σ^N and AD^N.

Normalized Signature Σ^N. Although the signature of $TeAL$ is very similar to that of AL, the use of prompts introduces additional system effects and constraints. In order to build a normalized system description, the signature of $TeAL$ must be extended to a normalized signature Σ^N. The sets of sorts, agents, and constants remain the same, but the sets of actions and fluents need to be changed. The normalized signature Σ^N is given by $\langle S, C, AG, FL^+, AC^N \rangle$, where FL^+ is the extended set of fluents, and AC^N is the set of prompts viewed as actions in the normalized action description AD^N.

Extended fluent sets FL^+. The use of prompts as new actions introduces additional preconditions and effects of these prompts. The **commence** prompts viewed as actions have common effects "*the action is in progress*" and "*the agent is executing the action,*" and common preconditions "*the action is not in progress*" and "*the agent is not executing any action.*" The **terminate** prompts have opposite effects and preconditions. I introduce two types of special fluents to F^+ to describe these common effects. They are of the form:

$$\mathbf{inProgress}\ Action \tag{4.23}$$

$$\mathbf{engaged}\ Ag \tag{4.24}$$

where $Action$ is an action and Ag is an agent. The fluent $\mathbf{inProgress}\ Action$ indicates that the action is being executed. For example:

$$\mathbf{inProgress}\ (serverA, nodeA)$$

means that the server is establishing a connection to the node. As will be discussed below, the logical value of $\mathbf{inProgress}\ Action$ becomes $true$ as the result of the commencing of the action, and becomes $false$ immediately after the action was terminated.

The second special fluent $\mathbf{engaged}\ Ag$ indicates that the agent Ag is executing an action. The logical value of $\mathbf{engaged}\ Ag$ is $true$ at the time moment after the agent commences any action and until the time moment when the action is terminated. In this

time interval Ag cannot commence any other actions. For example,

$$\textbf{engaged}\ serverA$$

means that the server is executing an action, which prevents it from commencing any other action before finishing the current one.

Thus, the fluent set FL^+ in Σ^N extends the fluent set FL in Σ by adding two sets of fluents: the **inProgress** fluents for all actions declared in Σ, and the **engaged** fluents for all agents declared in Σ.

$$FL^+ = FL \cup \{\textbf{inProgress}\ Action :\ Action \in AC\} \cup \{\textbf{engaged}\ Ag :\ Ag \in AG\}.$$

Thus, the fluents in FL^+ have three syntactic forms:

$$\textit{fluentName}(s_1, \ldots, s_k)$$
$$\textbf{inProgress}\ Action$$
$$\textbf{engaged}\ Ag$$

Normalized prompt set AC^N. In a normalized $TeAL$ theory prompts play the role of the actions in AL. For each action declaration statement:

$$\textbf{action}\ actionName(s_1, \ldots, s_k); \tag{4.25}$$

In a $TeAL$ theory Δ, we include in AC^N the prompt expressions:

$$\textbf{commence}\ actionName(s_1, \ldots, s_k).$$
$$\textbf{terminate}\ actionName(s_1, \ldots, s_k). \tag{4.26}$$

In AC^N, **commence** $Action$ represents the action of starting $Action$. Similarly, the prompt **terminate** $Action$ represents the action of terminating $Action$.

I also add a pair of *virtual* prompt expressions $totrue(F)$ and $tofalse(F)$ to AC^N, where F can be replaced by any fluent in FL^+. These two prompts indicate that the fluent F

is about to become $true$ or $false$, respectively. Their effects (F becomes $true$ or $false$) take place at the next time moment. The difference between virtual and regular prompts is that virtual prompts do not involve agents. Thus, the prompts $totrue$ and $tofalse$ can be used as prompts associated with changes in the system caused by other factors but prompts (such as passing time). In this way, every change in the system has a prompt associated with it. For instance, in the requirement "*a message becomes old if it has been received 10 minutes before*" there is no regular prompt that changes the fluent old to $true$, but $totrue(old)$ can be used in this case.

To recap, the action set AC^N is constructed as follows:

$$\begin{aligned} AC^N = & \{\mathbf{commence}\ Action:\ Action \in AC\} \\ & \cup\ \{\mathbf{terminate}\ Action:\ Action \in AC\} \\ & \cup\ \{totrue(P):\ P \in FL^+\} \cup \{tofalse(P):\ P \in FL^+\}. \end{aligned}$$

Normalized action description AD^N. To construct a normalized action description, I replace actions with prompts in the action language laws.

The three types of statements included in AD^N become:

State constraints	L **if** P	(4.27)
Dynamic causal laws	pr **causes** E **if** P	(4.28)
Executability conditions	**impossible** $pr_1, \ldots, pr_k$ **if** P	(4.29)

where E, L, P are lists of fluents and their negations, pr is a $\mathbf{terminate}$ prompt, and $pr_1, \ldots, pr_k$ are $\mathbf{commence}$ prompts. State constraints do not change because they do not involve actions. I replace the original actions in dynamic laws with $\mathbf{terminate}$ prompts because this expression specifies the effect of completing an action. I replace the actions in executability conditions with $\mathbf{commence}$ prompts because this expression specifies the cases when actions cannot start.

In addition, the use of prompts in $TeAL$ requires the normalized $TeAL$ theory to include additional statements representing the preconditions and effects that are applied to all prompts.

Intuitively, starting an action $Action$ (that is, executing **commence** $Action$) requires that **inProgress** $Action$ be false (negation be true) and results in **inProgress** $Action$ being true. Similarly, terminating $Action$ (that is, executing **terminate** $Action$) requires that the fluent **inProgress** $Action$ be true and results in **inProgress** $Action$ being false. I assume that no agent can perform multiple actions at the same time. The corresponding constraint is that an action cannot be commenced by an agent Ag if **engaged** Ag is true. The following expressions specify the constraints above. Because these expressions apply to all actions, the task of adding these expressions can be easily automated.

1. For every action $Action$ in AC, add the following *dynamic causal law* to AD^N:

$$\textbf{commence } Action \textbf{ causes inProgress } Action;$$

$$\textbf{terminate } Action \textbf{ causes not inProgress } Action;$$

$$\textbf{terminate } Action \textbf{ causes not engaged } Ag;$$

2. For every action $Action$ in AC, add the following *executability condition* to AD^N:

$$\textbf{impossible commence } Action \textbf{ if inProgress } Action;$$

$$\textbf{impossible terminate } Action \textbf{ if not inProgress } Action;$$

$$\textbf{impossible commence } Action \textbf{ if engaged } Ag;$$

There are also statements that need to be added for $totrue(F)$ and $tofalse(F)$. These statements specify the effect of these two types of prompts.

$$totrue(F) \textbf{ causes } F;$$

$$tofalse(F) \textbf{ causes not } F;$$

Transition Graph

Given a normalized $TeAL$ theory Δ^N, its signature $\Sigma(\Delta^N)$ and action description $AD(\Delta^N)$ form an action language AL theory $\langle \Sigma(\Delta^N), AD(\Delta^N) \rangle$. I define the semantics of this AL theory as a transition graph following the approach developed for AL by Baral and Gelfond [9] .

Complete and consistent sets of (possibly negated) fluents in FL^+ describe the state of the system. A transition is a triple $\langle s, prs, s' \rangle$, where s is the origin state, s' is the goal state, and prs is a set of prompts (they play the role of actions in AL). $TeAL$ allows concurrent prompts, thus prs can represent one prompt or multiple prompts that cause the state of the system to change from s to s'. The arc between s and s' is labeled with prs. The preconditions of every prompt in prs are satisfied by s and the effects of every prompts in prs are satisfied in s'. The only difference from the transition graph of Baral and Gelfond [9] is that, in $TeAL$, I use prompts in the role of actions.

Formally, given a $TeAL$ theory Δ, the transition graph described by its normalized theory Δ^N, T_{Δ^N}, is the pair $\langle S, R \rangle$, where:

1. S is the set of all states over FL^+ defined in the Δ^N such that, for every state constraint "L **if** P in Δ^N," a state s in S satisfies L if s satisfies P.

2. R is the set of triples $\langle s, prs, s' \rangle$ where prs represents a set of prompts, such that:

   – for every dynamic causal law "pr **causes** E **if** P" in Δ^N, if the state s satisfies P and the prompt set prs contains pr, then the state s' satisfies E.

   – for every executability condition "**impossible** $pr_1, \ldots, pr_k$ **if** P," if s satisfies P, then for every i such that $1 \leq i \leq k$, $pr_i \notin prs$.

For instance, assuming that Δ^N contains the expressions:

$$\textbf{terminate}\ send(sysA, msg, sysC)\ \ \textbf{causes}\ \ sent(sysA, msg, sysC)$$

$$\textbf{terminate}\ send(sysB, msg, sysC)\ \ \textbf{causes}\ \ sent(sysB, msg, sysC)$$

then for any transition $\langle s, prs, s' \rangle$ in T_{Δ^N}, if prs contains

$$\textbf{terminate } send(sysA, msg, sysC)$$

$$\textbf{terminate } send(sysB, msg, sysC)$$

then the two fluents that represent the effects

$$sent(sysA, msg, sysC)$$

$$sent(sysB, msg, sysC)$$

must hold in s'. For another example, let us take the expression:

$$\textbf{impossible commence } receive(sysC, msg, sysA) \textbf{ if not } sent(sysA, msg, sysC);$$

in AD^N. Then for any transition $\langle s, prs, s' \rangle$ in T_{Δ^N}, if prs contains the prompt **commence** $receive(sysC, msg, sysA)$, then $sent(sysA, msg, sysC)$ must hold in s.

Path and Timed Path

Paths in the transition graph T_{Δ^N} represent ways in which the system described by Δ^N may evolve. Questions such as whether there is a plan (sequence of actions) that transforms the system from a given state s to another state s'' can be stated as questions about the existence of a path from s to s'' in the transition graph T_{Δ^N}.

I define a path in T_{Δ^N} to be a sequence

$$\langle s_0, pr_0; s_1, pr_1; \ldots; s_{k-1}, pr_{k-1}; s_k \rangle$$

that satisfies the following conditions:

- $s_0 \ldots, s_k$ are states.
- $pr_0, \ldots, pr_{k-1}$ are sets of prompts.
- For every initial expression "**initially** F," the state s_0 satisfies F.

– For each $i = 0, \ldots, k-1$, $\langle s_i, pr_i, s_{i+1} \rangle$ is an edge in T_{Δ^N}.

The temporal constraints and duration specification expressions can be interpreted as constraints on the times of occurrences of prompts. To take the temporal aspects of Δ^N into account, I define a *timed path* p as a sequence:

$$\langle 0; s_0, pr_0, t_0; s_1, pr_1, t_1; \ldots; s_{k-1}, pr_{k-1}, t_{k-1}; s_k, t_k \rangle \tag{4.30}$$

where $\langle s_0, pr_0; s_1, pr_1; \ldots; s_{k-1}, pr_{k-1}; s_k \rangle$ is a path in T_{Δ^N} and $0 \leq t_0 < t_1 < \ldots < t_k$. This sequence starts from the initial state s_0. The prompts in pr_i take place at time moments t_i and the system is in state s_{i+1} during the time period $(t_i, t_{i+1}]$. The next set of prompts happens at time moment t_{i+1}. The first time moment of p is 0. The time t_k is the $horizon$ of p. I denote it as $h(p)$. We analyze the system in the interval $[0, t_k]$. I assume that all time parameters t_i, $0 \leq i \leq k$, are normalized to the same time unit and are integers.

Below is an example of a path based on the transition graph derived from the requirements introduced in Chapter 2. The fluents involved in this example are:

$$\textbf{inProgress}\ send(sysC, msg, sysA),\ \textbf{engaged}\ sysC,$$
$$\textbf{inProgress}\ receive(sysA, msg, sysC),\ \textbf{engaged}\ sysA,$$
$$\textbf{inProgress}\ send(sysB, msg, sysA),\ \textbf{engaged}\ sysB,$$
$$\textbf{inProgress}\ write(sysA, dev),\ sent(sysC, msg, sysA)$$

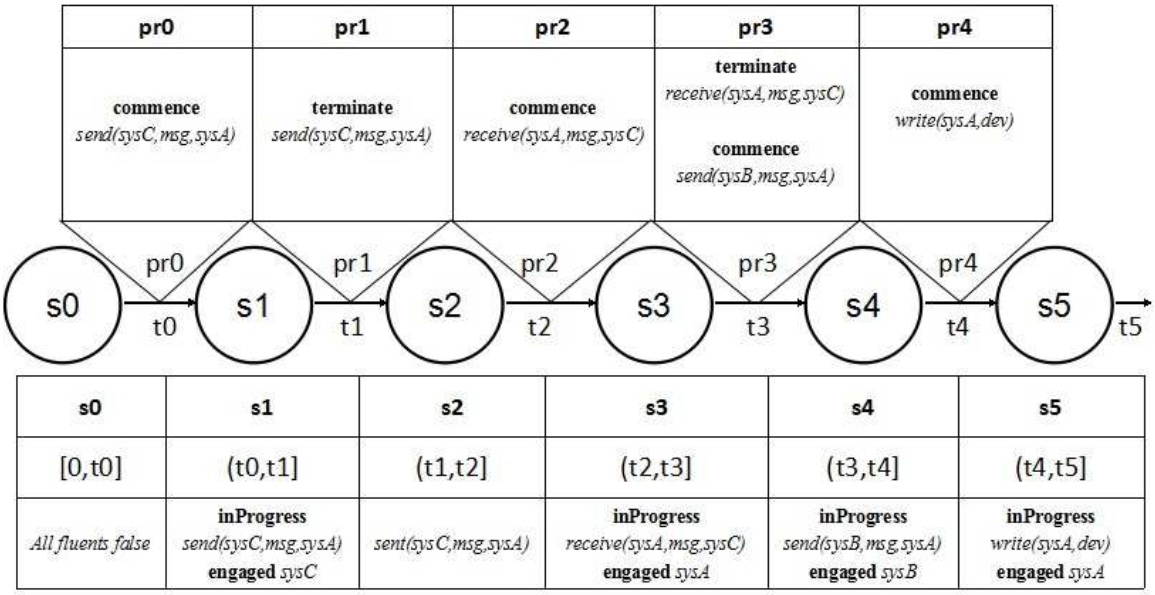
The atoms presented for each state are the fluents that are true in the state. The fluents that do not appear are false in the state. The path is given by:

$$\langle 0; s_0, pr_0, t_0; s_1, pr_1, t_1; \ldots; s_4, pr_4, t_4; s_5, t_5 \rangle \tag{4.31}$$

where the states and prompts are as follows:

– s_0: All fluents are false

– pr_0: **commence** $send(sysC, msg, sysA)$



s0	s1	s2	s3	s4	s5
[0,t0]	(t0,t1]	(t1,t2]	(t2,t3]	(t3,t4]	(t4,t5]
All fluents false	**inProgress** *send(sysC,msg,sysA)* **engaged** *sysC*	*sent(sysC,msg,sysA)*	**inProgress** *receive(sysA,msg,sysC)* **engaged** *sysA*	**inProgress** *send(sysB,msg,sysA)* **engaged** *sysB*	**inProgress** *write(sysA,dev)* **engaged** *sysA*

Figure 4.1: Example of transition graph

- s_1: **inProgress** $send(sysC, msg, sysA)$, **engaged** $sysC$
- pr_1: **terminate** $send(sysC, msg, sysA)$
- s_2: $sent(sysC, msg, sysA)$
- pr_2: **commence** $receive(sysA, msg, sysC)$
- s_3: **inProgress** $receive(sysA, msg, sysC)$, **engaged** $sysA$
- pr_3: **terminate** $receive(sysA, msg, sysC)$, **commence** $send(sysB, msg, sysA)$
- s_4: **inProgress** $send(sysB, msg, sysA)$, **engaged** $sysB$
- pr_4: **commence** $write(sysA, dev)$
- s_5: **inProgress** $write(sysA, dev)$, **engaged** $sysA$

This path represents a scenario that involves six states and five transitions: system C sends a message to system A, system A receives the message and begins to write it to the device. System B sends a message to system A at the same time when system A receives the

message. We can make it a timed path by labeling each prompt with a time moment, which shows when these changes happen. Let us consider a labeling $\{t_0, t_1, t_2, t_3, t_4, t_5\}$, where t_5 = 20 seconds. Since t_5 represents the horizon, the last time moment in the scenario is the 20th second. This labeling must satisfy all temporal constraints to make the timed path valid. For instance, if the $send(sysC, msg, sysA)$ action starts at t_0 and ends at t_1, the value of $t_1 - t_0$ must be equal to the duration of $send(sysC, msg, sysA)$. We can also check if the temporal constraint "*Subsystem A must write to the hardware device within K seconds after receiving a message from Subsystem C*" is satisfied on this timed path by checking if $t_4 - t_3 \ \leq \ K$ holds. The time moments t_3 and t_4 represent when the events "*Subsystem A receives a message from Subsystem C*" and "*Subsystem A writes to the hardware device*" occur. Because of the temporal constraint, for a time path to be valid in T_{Δ^N}, we must have $t_4 - t_3 \ \leq \ K$.

It should be noted that different timed paths can be created for a path, representing the same order of events but different times when they occur.

Satisfaction of a Temporal Condition on a Timed Path

Let us consider a temporal condition

$$C \ = \ \alpha \ timeComp \ timeMod \ \beta$$

where α and β are prompts or fluents (β can be empty), $timeComp$ can be $\mathbf{earlierThan}$, $\mathbf{at}$, $\mathbf{laterThan}$, $\mathbf{noEarlierThan}$, $\mathbf{noLaterThan}$, $timeMod$ can be $X \ unit$ $\mathbf{before}$, $X \ unit$ $\mathbf{after}$, or empty.

If α is a prompt, then let $occur(\alpha, u)$ represent the statement "α has occurred at time u." If α is a fluent F or the negation of F, then let $occur(\alpha, u)$ represent the statement "$totrue(F)$ has occurred at time u" or "$tofalse(F)$ has occurred at time u." Given a timed path p with horizon $h(p)$, I will now define the relation $p, t \models occur(\alpha, u)$ meaning that $occur(\alpha, u)$ holds on p at time t, where $t \in [0, h(p)]$.

– If α is $prtSymb\ A$, then $p, t \models occur(\alpha, u)$ holds if there is $i \in [0, k-1]$ such that $t_i = u$ and $\alpha \in pr_i$. It is possible that t_0=0.

– If α is $prtSymb$ **previous** A, then $p, t \models occur(\alpha, u)$ holds if there is $i \in [0, k-1]$ such that $t_i < t$, $u = t_i$, $\alpha \in pr_i$, and there is no j such that $t_i < t_j < t$ and $\alpha \in pr_j$. It is possible that t_0=0.

– If α is $prtSymb$ **next** A, then $p, t \models occur(\alpha, u)$ holds if there is $i \in [0, k-1]$ such that $t < t_i$, $u = t_i$, $\alpha \in pr_i$, and there is no j such that $t < t_j < t_i$ and $\alpha \in pr_j$.

Next, I define the relation $p, t \models C$ to represent that "the temporal condition C holds on p at time t." Let us consider that $Time_\alpha(t)$ represents the set of time moments determined by α with respect to time moment t. The set of time moments $Time_\alpha(t)$ is defined as follows:

– $Time_\alpha(t) = \{u : p, t \models occur(\alpha, u)\}$.

According to the definition of $p, t \models occur(\alpha, u)$ above, if α is $prtSymb\ A$, then the elements in $Time_\alpha(t)$ do not depend on t. If α is $prtSymb$ **previous** A or $prtSymb$ **next** A, then there is only one element in $Time_\alpha(t)$ and it depends on t.

Let us consider that $Time_\beta(t)$ represents the set of time moments determined by β and $timeMod$ with respect to time moment t. The set of time moments $Time_\beta(t)$ is defined as follows:

– If β is empty and $timeMod$ = "$x\ unit$ **after**," then $Time_\beta(t)$ has one element: $t + x$, x time units after time t.

– If β is empty and $timeMod$ = "$x\ unit$ **before**," then $Time_\beta(t)$ has one element: $t - x$, x time units before time t.

– If β is not empty and $timeMod$ = "$x\ unit$ **after**," then $Time_\beta(t) = \{v + x : p, t \models occur(\beta, v)\}$

– If β is not empty and $timeMod$ = "x $unit$ **before**," then $Time_\beta(t)$ = $\{v - x : p, t \models occur(\beta, v)\}$

The elements in $Time_\beta(t)$ can be greater than the horizon $h(p)$ or smaller than the start time 0. If β is not empty, the elements in $Time_\beta(t)$ are based on the time moments determined by β. If β is empty, the element in $Time_\beta(t)$ is determined by the "*current time moment*," t.

The relation $timeComp$ represents the comparison between the elements in $Time_\alpha(t)$ (el_α) and the elements in $Time_\beta(t)$ (el_β). The relation is defined as follows:

– In the case that $timeComp$ = **at**, if "el_α=el_β," then el_α is in relation $timeComp$ with el_β.

– In the case that $timeComp$ = **laterThan**, if "el_α>el_β," then el_α is in relation $timeComp$ with el_β.

– In the case that $timeComp$ = **earlierThan**, if "el_α<el_β," then el_α is in relation $timeComp$ with el_β.

– In the case that $timeComp$ = **noEarlierThan**, if "$el_\alpha \geq el_\beta$," then el_α is in relation $timeComp$ with el_β.

– In the case that $timeComp$ = **noLaterThan**, if "$el_\alpha \leq el_\beta$," then el_α is in relation $timeComp$ with el_β.

Given $C = \alpha$ $timeComp$ $timeMod$ β, if β is a prompt without **previous** or **next**, or if β is empty or **startTime**, then the relation $p, t \models C$ is defined as follows:

– $p, t \models C$ if there is no element $el_\beta \in Time_\beta(t)$ such that $el_\beta \leq h(p)$, and neither horizon $h(p)$ nor any element in $Time_\alpha(t)$ is in relation $timeComp$ with el_β.

– $p, t \not\models C$ if there exist an element $el_\beta \in Time_\beta(t)$ such that $el_\beta \leq h(p)$, and neither horizon $h(p)$ nor any element in $Time_\alpha(t)$ is in relation $timeComp$ with el_β.

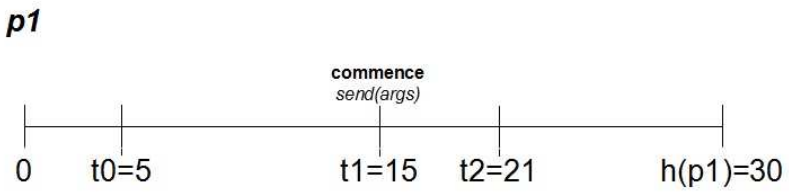


Figure 4.2: Example 1 of Satisfiability of Temporal Conditions

If β is a prompt with **previous** or **next**, then the relation $p, t \models C$ is defined as follows:

- $p, t \models C$ if $Time_\beta(t)$ is not empty, and there is no element $el_\beta \in Time_\beta(t)$ such that $el_\beta \leq h(p)$, and neither horizon $h(p)$ nor any element in $Time_\alpha(t)$ is in relation $timeComp$ with el_β.

- $p, t \not\models C$ if there does not exist any element in $Time_\beta(t)$, or there exist an element el_β in $Time_\beta(t)$ such that $el_\beta \leq h(p)$, and neither horizon $h(p)$ nor any element in $Time_\alpha(t)$ is in relation $timeComp$ with el_β.

It should be noted that given a timed path p, the occurrences of prompts are only determined within the horizon $h(p)$. If the satisfiability of a temporal condition C at time t requires a prompt occur (or not occur) after $h(p)$, then we consider C to be satisfied at t because we have no evidence that this prompt does not occur (or occurs) after $h(p)$. This concept will be illustrated in the examples below.

I will now illustrate the definition of $p, t \models C$ with examples. Given a temporal condition C:

$$\textbf{commence}\ send(\overline{args})\ \textbf{noLaterThan}\ 10\ second\ \textbf{after}$$

the prompt **commence** $send(\overline{args})$ is α, and β is empty. According to the definitions above,

$$Time_\alpha(t) = \{u : p, t \models occur(\textbf{commence}\ send(\overline{args}), u)\}$$
$$Time_\beta(t) = \{t + x\}$$

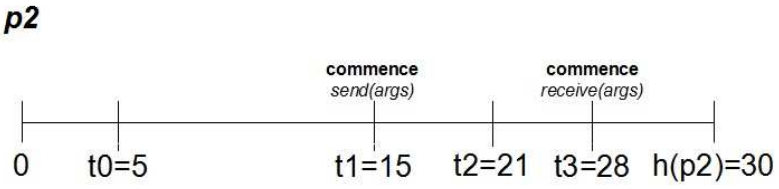


Figure 4.3: Example 2 of Satisfiability of Temporal Conditions

Let us consider a timed path p_1 (Figure 4.2), where the horizon $h(p_1) = 30$, and **commence** $send(\overline{args})$ occurs at 15. That is, $Time_\alpha(t)$ has one element el_α = 15 for any t on p_1. We have $p_1, t \models C$ for any $t \in [5, 30]$, because for any t in this interval, $el_\alpha \leq t+10$. However, $p_1, t \not\models C$ for any $t \in [0, 4]$. It should be noted that $p_1, t \models C$ for any $t \in [21, 30]$ even if there is no occurrence of **commence** $send(\overline{args})$ on p_1, because $h(p_1) \leq t + 10$. In this case the horizon is not large enough for us to assert that **commence** $send(\overline{args})$ does not occur early enough after the horizon. In other words, we consider C to be satisfied at any $t \in [21, 30]$ because we have no evidence that C is violated in this interval.

If we change the **commence** $send(\overline{args})$ in C to be **commence next** $send(\overline{args})$, then $p_1, t \models C$ in the interval $[5, 14]$ and $[21, 30]$ and $p_1, t \not\models C$ in the interval $[0, 4]$ and $[15, 20]$. The change of the satisfiability in $[15, 20]$ is because of the **next** keyword. For any $t \in [15, 20]$, $Time_\alpha(t)$ is empty because the next occurrence of **commence** $send(\overline{args})$ does not exist, and the horizon $h(p_1)$ does not satisfy $h(p_1) \leq t + 10$ either.

For another example, let C be:

commence $receive(\overline{args})$

noLaterThan 10 $second$ **after**

commence previous $send(\overline{args})$

the prompt **commence** $receive(\overline{args})$ is α, and β is **commence previous** $send(\overline{args})$. Let us consider a timed path p_2 (Figure 4.3), where the horizon $h(p_2) = 30$, **commence** $send(\overline{args})$ occurs at 15, and **commence** $receive(\overline{args})$ occurs at 28. That is, $Time_\alpha(t)$ has one element el_α = 28, $Time_\beta(t)$ has one element 15 for any $t \in [16, 30]$. We have

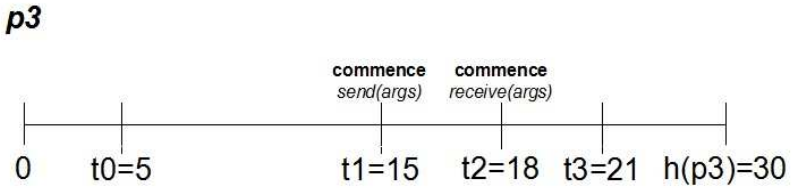


Figure 4.4: Example 3 of Satisfiability of Temporal Conditions

$p_2, t \models C$ for any $t \in [18, 30]$, because for any t in this interval, $el_\alpha \leq t + 10$. However, $p_2, t \not\models C$ for any $t \in [16, 17]$. We also have $p_2, t \not\models C$ for any $t \in [0, 15]$, because for these time moments the previous occurrence of **commence** $send(\overline{args})$ does not exist. Similar to the example above, $p_2, t \models C$ for any $t \in [21, 30]$ even if there is no occurrence of **commence** $receive(\overline{args})$ on p_2, because $h(p_2) \leq t + 10$.

The examples above show temporal conditions $C = \alpha\ timeComp\ timeMod\ \beta$ where β is empty or β involves **previous** or **next**. The satisfiability these examples above depends on t. That is, there may exist t' and $t''\in[0, h(p)]$ such that $p, t' \models C$ and $p, t'' \not\models C$. We call such temporal conditions *local*.

If we change the temporal condition C to be:

$$\textbf{commence next } receive(\overline{args})$$
$$\textbf{noLaterThan } 10\ second \textbf{ after}$$
$$\textbf{commence previous } send(\overline{args})$$

and change the timed path to p_3 (Figure 4.4), where **commence** $receive(\overline{args})$ occurs at 18, then $p_3, t \models C$ in the interval $[16, 17]$ and $p_3, t \not\models C$ in the interval $[18, 20]$. This is because the next occurrence of **commence** $receive(\overline{args})$ exists for the time moment in $[16, 17]$, but not for the time moments in $[18, 20]$.

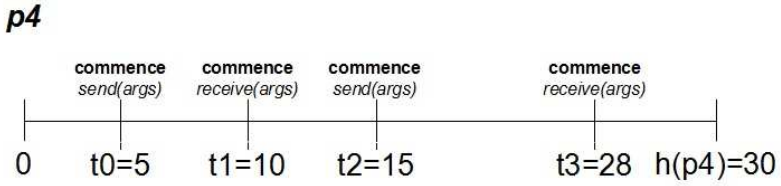


Figure 4.5: Example 4 of Satisfiability of Temporal Conditions

If we change the temporal condition C to be:

$$\textbf{commence}\ receive(\overline{args})$$
$$\textbf{noLaterThan}\ 10\ second\ \textbf{after}$$
$$\textbf{commence}\ send(\overline{args})$$

the prompt **commence** $receive(\overline{args})$ is α, and β is **commence** $send(\overline{args})$. Let us consider a timed path p_4 (Figure 4.5), where the horizon $h(p_4) = 30$, **commence** $send(\overline{args})$ occurs at 5 and 15, and **commence** $receive(\overline{args})$ occurs at 10 and 28. That is, $Time_\alpha(t)$ has two elements 10 and 28, $Time_\beta(t)$ has two element 5 and 15 for any $t \in [0, h(p_4)]$. We have $p_4 \models C$ because for both elements in $Time_\beta(t)$ there is el_α = 10 such that $el_\alpha \leq el_\beta + 10$.

If we change C to be:

$$\textbf{commence next}\ receive(\overline{args})$$
$$\textbf{noLaterThan}\ 10\ second\ \textbf{after}$$
$$\textbf{commence}\ send(\overline{args})$$

then $p_4 \not\models C$. This is because for the **commence** $send(\overline{args})$ occurs at 15, the next occurrence of **commence** $receive(\overline{args})$ is at 28, and $28 > 15 + 10$. Since $p_4, t2 \not\models C$, we have $p_4 \not\models C$.

The two examples above show temporal conditions $C = \alpha\ timeComp\ timeMod\ \beta$ where β is a prompt without **previous** or **next**. The satisfiability of these temporal conditions does not depend on t. That is, for any t' and $t''\in[0, h(p)]$, $p_4, t' \models C$ if and only if

$p, t'' \models C$, $p, t' \not\models C$ if and only if $p, t'' \not\models C$. We call such temporal conditions as *global* temporal conditions.

Consistency Checking

If $p, t \models C$ for every t ($0 \leq t \leq h(p)$), then C is *satisfied by* p or C *holds on* p, written as $p \models C$. Let $I = [0, h]$ represent the set of time moments from 0 to h. If a timed path p can be found in the transition graph T_{Δ^N} such that for every temporal constraint C in TC, $p \models C$, and $h(p) = h$, then the $TeAL$ theory is *consistent* on I.

Check Points. Let us recall that a valid path in T_{Δ^N} becomes a timed path when times are assigned to states. A timed path is valid if all temporal constraints are satisfied at every time moment on this timed path. However, checking the satisfiability at every time moment is infeasible. Instead, we can check the satisfiability of temporal conditions at a finite set of *check points* and reduce the task of checking satisfiability along a timed path to checking satisfiability at every check point.

I define two types of check points. The first type, *state determined check points*, comprise the time moments when the system changes state. The time when the system changes state is defined as the *last* time moment when the system is still in its present state. These are also the times when prompts occur. Additionally, the start time, time moment 0, is also regarded as a state defined check point. The second type of check points, *condition determined check points*, comprise the time moments when nothing changes in the system, but the *satisfaction of a temporal conditions* might change. Each temporal condition determines a group of condition determined check points.

Both types of check points are necessary for checking the satisfiability of temporal constraints on a timed path. For example, let us consider a timed path p_5 (Figure 4.6) such that $h(p_5) = 15$ and **commence** $send(\overline{args})$ occurs on p_5 at time moments $t_0 = 2$, $t_3 = 8$,

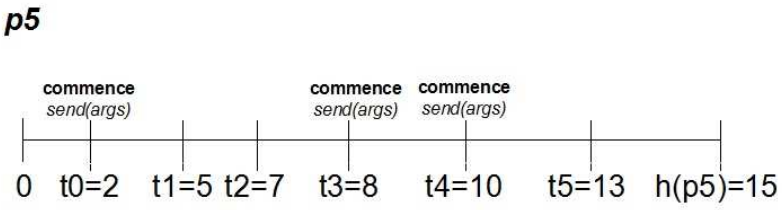


Figure 4.6: Example of Check Points

$t_4 = 10$. Given a temporal condition C:

$$\textbf{commence next } send(\overline{args}) \textbf{ noLaterThan } 3 \ second \textbf{ after}$$

the time moments when **commence** $send(\overline{args})$ occurs (t_0, t_3, t_4) and the start time 0 are state determined check points. Checking the satisfiability of C at these check points shows that C is satisfied on p_5 at 0 and t_3. However, C is violated at t_0 and t_4. The condition determined check points for C are "3 seconds before the time moments when **commence** $send(\overline{args})$ occurs" ($t_1 = 5$, $t_2 = 7$) and "2 seconds before the horizon" ($t_5 = 13$). Checking the satisfiability of C at these check points shows that C is satisfied at t_1, t_2, and t_5. The results of both types of check points show that C is satisfied on p_5 at any time moments in $[0, t_0)$, $[t_1, t_4)$, $[t_5, h(p_5)]$, and violated at any time moments in $[t_0, t_1)$, $[t_4, t_5)$. If we do not check the satisfiability of C at the state determined check points t_0 and t_4, we will not find that C is violated in $[t_0, t_1)$, $[t_4, t_5)$.

If we change C to be:

$$\textbf{commence next } send(\overline{args}) \textbf{ laterThan } 3 \ second \textbf{ after}$$

then C is violated in $[0, t_0)$ and $[t_1, t_4)$. If we do not check the satisfiability of C at the condition determined check points t_1, then we will not find that C is violated in $[t_1, t_4)$. If we do not check the satisfiability of C at the state determined check points 0, then we will not find that C is violated in $[0, t_0)$.

I will now provide a precise definition of check points of both types. Let us consider that C represents a temporal condition $\alpha \ timeComp \ timeMod \ \beta$. The value of $timeComp$

can be **earlierThan**, **at**, **laterThan**, **noEarlierThan**, and **noLaterThan**. The value of $timeMod$ can be x $unit$ **before**, x $unit$ **after**. The $timeMod$ can be empty. Take a timed path *p*, I denote the set of check points on *p* with regard to *C* as:

$$CP(C,p) \;=\; state(C,p) \;\cup\; condition(C,p)$$

where the notation $state(C,p)$ represents the set of state determined check points, and $condition(C,p)$ represents the set of condition determined check points.

I denote the sets of all times when α and β occur on p as T_α and T_β, respectively. The set $state(C,p)$ includes T_α, T_β, and 0, the start time of the system.

$$state(C,p) \;=\; T_\alpha \;\cup\; T_\beta \;\cup\; \{0\}$$

The definition of $condition(C,p)$ depends on the values of α, β and $timeComp$ $timeMod$. Given a timed path p with horizon $h(p)$, we use the following definition to represent the set of time moments that is x seconds after or before the times when α occurs ($x \geq 0$).

$$T_\alpha + x \;=\; \{t_i \;+\; x \mid t_i \;+\; x \leq h(p),\; t_i \;\in\; T_\alpha\}.$$
$$T_\alpha - x \;=\; \{t_i \;+\; x \mid t_i \;-\; x \geq 0,\; t_i \;\in\; T_\alpha\}.$$

The condition determined check points introduced because of α are:

– all elements of $T_\alpha + 1$ if $\alpha = prtSymb$ **previous** A.

If $\alpha = prtSymb$ **previous** A, then for every $t \in [t_j + 1, t_{j+1} + 1)$ where $t_j, t_{j+1} \in T_\alpha$, the previous occurrence of $prtSymb$ A, α, indicates the same time moment t_j. Similarly, if β is not empty, the condition determined check points introduced because of β are:

– all elements of $T_\beta + 1$, if $\beta = prtSymb$ **previous** B.

If β is empty, the condition determined check points introduced because of $timeComp$ and $timeMod$ are given as following:

– if $timeMod$ = x $unit$ **after**, $timeComp$ = **earlierThan** or **noEarlierThan**, then the condition determined check points are all elements of ($T_\alpha - (x-1)$).

– if $timeMod$ = x $unit$ **after**, $timeComp$ = **laterThan**, then the condition determined check points are all elements of ($T_\alpha - x$).

– if $timeMod$ = x $unit$ **after**, $timeComp$ = **noLaterThan**, then the condition determined check points are all elements of ($T_\alpha - x$).

– if $timeMod$ = x $unit$ **after**, $timeComp$ = **at**, then the condition determined check points are all elements of ($T_\alpha - x$) $\cup$ ($T_\alpha - (x-1)$).

– if $timeMod$ = x $unit$ **before**, $timeComp$ = **earlierThan** or **noEarlierThan**, then the condition determined check points are all elements of ($T_\alpha + (x+1)$).

– if $timeMod$ = x $unit$ **before**, $timeComp$ = **laterThan** or **noLaterThan**, then the condition determined check points are all elements of ($T_\alpha + x$).

– if $timeMod$ = x $unit$ **before**, $timeComp$ = **at**, then the condition determined check points are all elements of ($T_\alpha + x$) $\cup$ ($T_\alpha + (x+1)$).

Theorem 1 *Let p be a timed path, $C = \alpha$ $timeComp$ $timeMod$ β be a temporal condition, and $CP(C,p) = \{c_0, \ldots, c_k\}$ where $c_0 < \ldots < c_k$.*

– *For every i, $0 \leq i \leq k-1$, if $p, c_i \models C$, then for every $t \in [c_i, c_{i+1})$, $p, t \models C$.*

Proof. Given a temporal condition $C = \alpha$ $timeComp$ $timeMod$ β, the satisfiability of C on a timed path p at time t is determined by the relation between two sets of time moments $Time_\alpha(t)$, which is determined by α, and $Time_\beta(t)$, which is determined by $timeComp$, $timeMod$ and β.

If there is a time moment $dif \in [c_i, c_{i+1})$ such that the satisfiability of C at dif is different from the satisfiability at c_i, then either $Time_\alpha(dif)$, the set of time moments

determined by α at time dif, is different from $Time_\alpha(c_i)$, or $Time_\beta(dif)$ is different from $Time_\beta(c_i)$.

If $\alpha = prtSymb$ **previous** A and $c_i \in T_\alpha$, then $[c_i, c_{i+1})$ only has one element c_i because $condition(C, p)$ includes the set of time moments $T_\alpha + 1$, and $c_i + 1$ is an element of $T_\alpha + 1$. It is impossible that the satisfiability at dif and c_i is different because of α. This is a contradiction.

If $\alpha = prtSymb$ **previous** A and $c_i \notin T_\alpha$, then $Time_\alpha(dif)$ includes at most one element c_j such that $c_j < dif$, $c_j \in T_\alpha$, and there is no c_k such that $c_j < c_k < dif$ and $c_k \in T_\alpha$. This element c_j represents the latest time moment before dif when $prtSymb\ A$ occurs. If $Time_\alpha(dif)$ is empty, then $prtSymb\ A$ does not occur before dif. In this case, $prtSymb\ A$ does not occur before c_i as well, and $Time_\alpha(c_i)$ is also empty. Similarly, $Time_\alpha(c_i)$ includes at most one element c'_j such that $c'_j < c_i$, $c'_j \in T_\alpha$, and there is no c'_k such that $c'_j < c'_k < c_i$ and $c'_k \in T_\alpha$. This element c'_j represents the latest time moment before c_i when $prtSymb\ A$ occurs. If $Time_\alpha(c_i)$ is empty, then because $dif \in [c_i, c_{i+1})$ and $c_i \notin T_\alpha$, $Time_\alpha(dif)$ is empty as well. If both $Time_\alpha(dif)$ and $Time_\alpha(c_i)$ are not empty, and c_j and c'_j are different, then there must be $c_i < c_j < dif < c_{i+1}$, which means that a check point c_j exists between c_i and c_{i+1}. This is a contradiction.

If $\alpha = prtSymb$ **next** A, then $Time_\alpha(dif)$ includes at most one element c_j such that $c_j > dif$, $c_j \in T_\alpha$, and there is no c_k such that $c_j > c_k > dif$ and $c_k \in T_\alpha$. This element c_j represents the earliest time moment after dif when $prtSymb\ A$ occurs. Similarly, $Time_\alpha(c_i)$ includes at most one element c'_j such that $c'_j > c_i$, $c'_j \in T_\alpha$, and there is no c'_k such that $c'_j > c'_k > c_i$ and $c'_k \in T_\alpha$. This element c'_j represents the earliest time moment after c_i when $prtSymb\ A$ occurs. If c_j and c'_j are different, then there must be $c_i \leq dif < c_j < c_{i+1}$, which means that a check point c_j exists between c_i and c_{i+1}. This is a contradiction.

If $\alpha = prtSymb\ A$, then the elements of $Time_\alpha(dif)$ and $Time_\alpha(c_i)$ are the same. It is impossible that the satisfiability at t and c_i is different because of α. This is a contradiction.

The discussion above also applies to the cases that $\beta = prtSymb$ **previous** B, $\beta = prtSymb$ **next** B, and $\beta = prtSymb\ B$.

If β is empty, the condition determined check points are introduced based on $timeComp$ and $timeMod$.

If $timeComp$ is **at** and $timeMod$ is $x\ second$ **after**, then the check points divide the timed path into four types of intervals.

- The first type of intervals is $[0, c^f_\alpha - x)$, where c^f_α is the smallest element of T_α such that $c^f_\alpha > x$. For any time t in this interval, the temporal condition is violated.

- The second type of intervals is $[h(p) - x + 1, h(p)]$, where $h(p)$ is the horizon of the timed path p. For any time t in this interval, the temporal condition is satisfied.

- The third type of intervals is $[c^i_\alpha - x, c^i_\alpha - x + 1)$, where c^i_α can be any element in T_α such that $c^i_\alpha > x$. It is impossible to find a time dif because these types of intervals have only one element $c^i_\alpha - x$.

- The forth type of intervals is $[c^i_\alpha - x + 1, c^j_\alpha - x)$, where c^j_α is the smallest element in T_α such that $c^j_\alpha > c^i_\alpha$. For any time t in these intervals C is violated.

It is impossible to find a time dif in these types of intervals.

If $timeComp$ is **noLaterThan** and $timeMod$ is $x\ second$ **after**, then the check points divide the timed path into four types of intervals. The first two types are the same as above. It is impossible to find a time dif in these two types of intervals. The third type of intervals is $[c^i_\alpha - x, c^i_\alpha)$, where c^i_α can be any element in T_α such that $c^i_\alpha > x$. For any time t in these intervals C is satisfied. The forth type of intervals is $[c^i_\alpha, c^j_\alpha - x)$, where c^j_α is the smallest element in T_α such that $c^j_\alpha > c^i_\alpha + x$. For any time t in these intervals C is violated. It is impossible to find a time dif in these two types of intervals.

If $timeComp$ is **laterThan** and $timeMod$ is $x\ second$ **after**, then the check points divide the timed path into four types of intervals like the case in which $timeComp$ is

noLaterThan and *timeMod* is x *second* **after**. The only difference is that the second type is $[h(p) - x + 2, h(p)]$ instead of $[h(p) - x + 1, h(p)]$. The temporal condition is satisfied at every time moment in the first, second and forth types of intervals, and is violated at every time moment in the third type of intervals.

If *timeComp* is **noEarlierThan** and *timeMod* is x *second* **after**, then the check points divide the timed path into four types of intervals. The first two types are the same as above. The difference is that the first type is $[0, c^f_\alpha - x + 1)$ instead of $[0, c^f_\alpha - x)$. It is impossible to find a time *dif* in these two types of intervals. The third type of intervals is $[c^i_\alpha - x + 1, c^i_\alpha)$, where c^i_α can be any element in T_α such that $c^i_\alpha > x - 1$. For any time t in these intervals C is violated. The forth type of intervals is $[c^i_\alpha, c^j_\alpha - x + 1)$, where c^j_α is the smallest element in T_α such that $c^j_\alpha > c^i_\alpha + x - 1$. For any time t in these intervals C is satisfied.

If *timeComp* is **earlierThan** and *timeMod* is x *second* **after**, then the check points divide the timed path into four types of intervals like the case in which *timeComp* is **noEarlierThan** and *timeMod* is x *second* **after**. The temporal condition is satisfied at every time moment in the second and third types of intervals, and is violated at every time moment in the first and forth type of intervals.

If *timeComp* is **at** and *timeMod* is x *second* **before**, then the check points divide the timed path into four types of intervals.

- The first type of intervals is $[0, c^f_\alpha + x)$, where c^f_α is the smallest element of T_α. For any time t in this interval, the temporal condition is violated.

- The second type of intervals is $[c^e_\alpha + x + 1, h(p)]$, where $h(p)$ is the horizon of the timed path p and c^e_α is the largest element of T_α such that $c^e_\alpha < h(p) - x$. For any time t in this interval, the temporal condition is violated.

- The third type of intervals is $[c^i_\alpha + x, c^i_\alpha + x + 1)$, where c^i_α can be any element in T_α such that $c^i_\alpha < h(p) - x$. It is impossible to find a time *dif* because these types of

intervals have only one element $c_\alpha^i + x$.

– The forth type of intervals is $[c_\alpha^i + x + 1, c_\alpha^j + x)$, where c_α^j is the smallest element in T_α such that $c_\alpha^j > c_\alpha^i$ and $c_\alpha^j < h(p) - x$. For any time t in these intervals C is violated.

It is impossible to find a time dif in these types of intervals.

In the cases that $timeMod$ is x $second$ **before** and $timeComp$ is **noLaterThan**, **laterThan**, **noEarlierThan** or **earlierThan**, the check points divide the timed path into four types of intervals like the cases above. For each interval, the temporal condition is either satisfied or violated on the entire interval.

Based on the discussion above, for any temporal condition C, $CP(C, p)$ satisfied the properties:

– for every i, $0 \leq i \leq k-1$, if $p, c_i \models C$, then for every $t \in [c_i, c_{i+1})$, $p, t \models C$.

– for every i, $0 \leq i \leq k-1$, if $p, c_i \not\models C$, then for every $t \in [c_i, c_{i+1})$, $p, t \not\models C$.

That is, $CP(C, p)$ includes sufficient check points. Theorem 1 is proved. □

Corollary 2 *Let p be a timed path, $C = \alpha\ timeComp\ timeMod\ \beta$ be a temporal condition, and $CP(C, p) = \{c_0, \ldots, c_k\}$ where $c_0 < \ldots < c_k$. If for every i, $0 \leq i \leq k$, there is $p, c_i \models C$, then $p \models C$.*

Chapter 5 $TeALGenerator$

The approach introduced in this thesis consists of two phases: generating a $TeAL$ theory based on the requirements given in natural language, and checking the consistency of the requirements based on this $TeAL$ theory. The analysts' involvement is only necessary in the first phase. This phase consists of four steps:

1. Extracting relevant requirements
2. Identifying system elements
3. Constructing "close-to-$TeAL$" statements ($AlmostTeAL$)
4. Converting $AlmostTeAL$ statements to correct $TeAL$ statements

I designed and implemented a tool, $TeALGenerator$, to automate the process of generating $AlmostTeAL$ statements from natural language requirements. The tool uses my code and Natural Language Processing (NLP) tools including Stanford Parser [52, 24] and the semantic role labeler Senna [21]. The $TeALGenerator$ assumes that all requirements that are necessary for modeling the system have been found in step 1, and performs steps 2 and 3 automatically. The output of $TeALGenerator$ is $AlmostTeAL$ statements that analysts need to verify in step 4. High quality $AlmostTeAL$ statements make the task easier.

5.1 Data Flow of $TeALGenerator$

I assume that the input of $TeALGenerator$ is a set of text files, with each file containing a natural language requirement. The $TeALGenerator$ first processes each file with Senna and the Stanford Parser (the outcomes of these tools are described below). Then $TeALGenerator$ identifies system concepts from the outcomes of Senna and Stanford Parser. The system concepts include actions, fluents, temporal constraints, non-temporal

constraints, and relations among the constraints. $TeALGenerator$ uses such information to generate $AlmostTeAL$ statements. For example, given a requirement "*If the system is not in safe mode or a message is received in last 10 second, ...*," the $TeALGenerator$ generates the $AlmostTeAL$ expression:

$$\begin{aligned}&\textbf{if not } in(system, safeMode) \textbf{ or}\\ &\qquad receive(\_, message) \textbf{ noEarlierThan } 10\ second \textbf{ before}\\ &\qquad \textbf{then } \ldots;\end{aligned}$$

The action $receive(\_, message)$ is identified using Senna; the fluent $in(system, safeMode)$, the temporal constraint $\textbf{noEarlierThan}\ 10\ second\ \textbf{before}$, the non-temporal constraint $\textbf{if}\ \ldots$, the disjunction relationship $\textbf{or}$, and the negation $\textbf{not}$, are identified using the Stanford Parser. The details are discussed below.

Output of Senna

The output of Senna illustrates the predicates in the sentence, the semantic arguments associated with them, and the roles of these arguments. The predicates can be used as action names in $AlmostTeAL$. The roles of arguments represent the argument types. Actions such as $update(system, data)$ and some fluents, such as $received(receiver, msg, sender)$, can be identified in this way.

For instance, Senna processes the text "*If the system is not in safe mode or a message is received in last 10 second*" and generates the results in Table 5.1.

The first column shows the original text. The second column shows the Parts-Of-Speech (POS) [84] tags for each word. The tags in this example include: IN (preposition or subordinating conjunction), DT (determiner), NN (noun), VBZ (3rd person singular present), RB (adverb), JJ (adjective), CC (coordinating conjunction), VBN (verb, past participle), CD (cardinal number), and NNS (noun, plural). The third column shows the chunks, or short phrases, in the input text. The tags in this column are the phrase level

Table 5.1: Output of Senna

Text	POS	Chunk	Predicate	received Args
If	IN	S-SBAR		
the	DT	B-NP		
system	NN	E-NP		
is	VBZ	S-VP		
not	RB	O		
in	IN	S-PP		
safe	JJ	B-NP		
mode	NN	E-NP		
and	CC	O		
a	DT	B-NP		B-A1
message	NN	E-NP		E-A1
is	VBZ	B-VP		
received	VBN	E-VP	received	S-V
in	IN	S-PP		B-AM-TMP
last	JJ	B-NP		I-AM-TMP
10	CD	I-NP		I-AM-TMP
seconds	NNS	E-NP		E-AM-TMP

tags used in the Penn Treebank Project [65]. The tags in this example include: $SBAR$ (a clause introduced by a possibly empty subordinating conjunction), S (simple declarative clause), NP (proper noun, singular), VP (proper verb), and PP (prepositional phrase). The prefixes of the tags (B and E) represent the beginning and the end of phrases. The POS tags and chunks can be used to determine if a predicate is an action or a fluent. The fourth column shows the predicates Senna found in this requirement. In our example, this column only has one value, for there is only one predicate identified: $received$. The fifth column shows the two arguments Senna identified for $received$. Senna uses the tags $A1$ and AM_TMP to represent the type of the arguments. For the predicate $received$, $A1$ marks the argument that represents the received object (*a message*), AM_TMP marks the argument that represents the time when $received$ happened (*in last 10 seconds*). Senna uses the tag $A0$ for agents. The tag $A0$ does not appear in Table 5.1 because the sample text does not specify the agent that receives the message.

The $TeALGenerator$ reads the Senna outputs generated for all requirements and saves

all information in a $sennaInfo$ object. The algorithm for reading the output of Senna is given as Algorithm 1.

Algorithm 1 Read the Senna output

Data: output of Senna, sennaInfo

Result: sennaInfo

foreach *word w* ***in*** *the text processed by Senna* **do**

  if *w is a predicate* **then**

    foreach *argument arg of w* **do**

      get its text, role, position in the sentence, pos tag, chunk tag;

      save all information to sennaInfo;

    end

  end

end

Output of Stanford Parser

The Stanford Parser generates a parse tree and a list of Stanford Dependencies (**SD**) for each sentence in the given text. Each parse tree has a $ROOT$ node. For example, Figure 5.1 shows the subtree that corresponds to the text "*If the system is not in safe mode or a message is received in last 10 second*:" This figure shows the tags (NN, VBZ, ...) for the words, and the Penn Treebank tags ($SBAR$, VP, ...) for the phrases. Every subtree in Figure 5.1 represents a phrase. For example, the subtree $(NP\ (JJ\ safe)\ (NN\ mode))$ (the subtree rooted in the node "7. NP" in Figure 5.1) represents "$safe\ mode$."

A Stanford dependency is given in the form of $reln(gov, dep)$. It represents that a grammatical relation ($reln$) holds between a governor (gov) and a dependent (dep). For example, $nsubj(is, system)$ means that $system$ is the syntactic subject ($nsubj$) of a clause, and $prep\_in(is, mode)$ means that in is a prepositional modifier ($prep\_in$) of is. Table 5.2 includes the fifteen typed dependencies identified in the text "*If the system is not in safe mode or a message is received in last 10 second*." The numbers after the governor and the

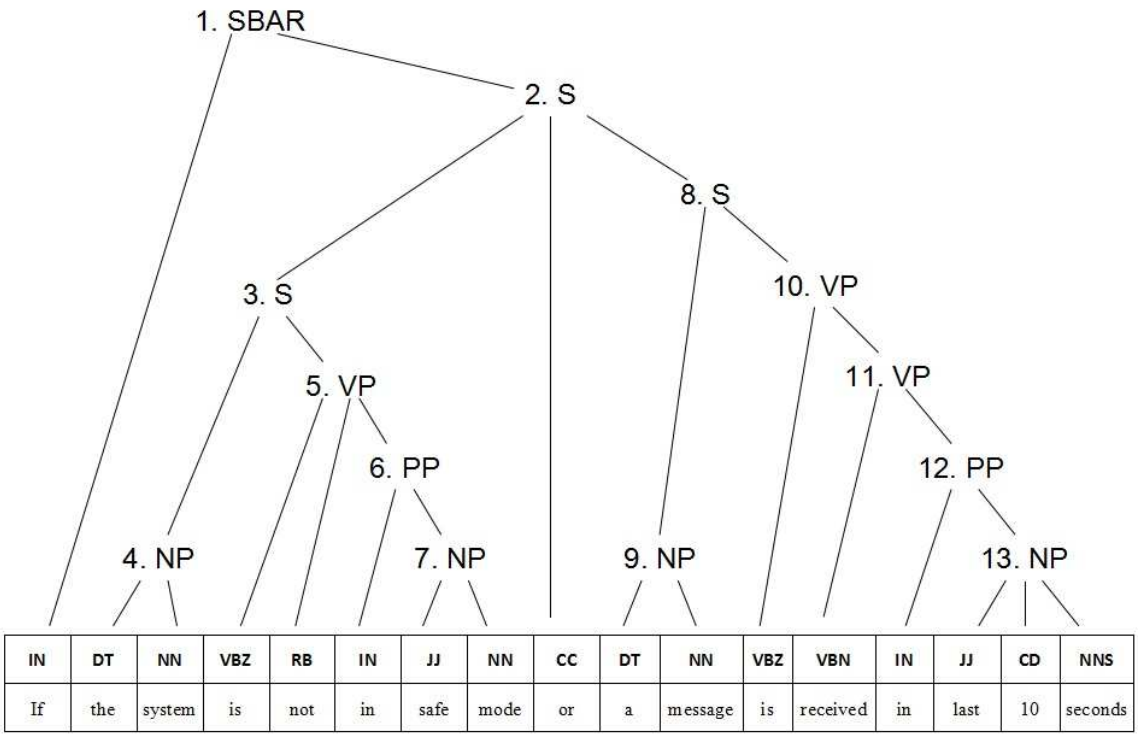


Figure 5.1: Example of Stanford Parse Tree

dependent show the position of these words in the text. The detailed definition of these relations can be found in the Stanford Dependencies Manual [25].

Identifying System Information

The $TeALGenerator$ identifies two types of system elements: *vocabulary* and *constraints*, based on the outputs of Senna and Stanford Parser. The vocabulary refers to the constants of the system and information about actions and fluents, including their names and arguments. These types of information are used for constructing the signature Σ and the action description in $TeAL$. Constraints refer to the non-temporal constraints in AD and the temporal constraints in TC.

Identifying Vocabulary

The first task of $TeALGenerator$ is to identify the predicates in the input requirements. These predicates will be used as actions and fluents in $AlmostTeAL$ statements. The

Table 5.2: Stanford Dependencies

ID	Relation	Governor	Dependent
1	mark	is-4	If-1
2	det	system-3	the-2
3	nsubj	is-4	system-3
4	advcl	report-22	is-4
5	neg	is-4	not-5
6	amod	mode-8	safe-7
7	prep_in	is-4	mode-8
8	det	message-11	a-10
9	nsubjpass	received-13	message-11
10	auxpass	received-13	is-12
11	conj_and	is-4	received-13
12	advcl	report-22	received-13
13	amod	seconds-17	last-15
14	num	seconds-17	10-16
15	prep_in	received-13	seconds-17

$TeALGenerator$ uses the output of Senna and the Stanford Parser to identify the predicate names and arguments.

As mentioned earlier, the output of Senna identifies the predicates in the input text, the semantic arguments associated with these predicates, and the roles of these arguments. In the example text "*If the system is not in safe mode or a message is received in last 10 seconds*," Senna identifies the predicate $received$ and its two arguments: *a message* and *in last 10 seconds*.

If the Parts-Of-Speech tag of the predicate word is VB ($receive$), VBZ ($receives$), or VBG ($receiving$), the predicate is considered as an action. Otherwise, if the chunk that contains this word is a noun phrase (*a received message*), the predicate is considered as a fluent. If the chunk is a verb phrase (*is received*), then the predicate is still considered as an action. In the sample text, the $received$ predicate (node 3 in Figure 5.3) is considered as an action. The $TeALGenerator$ applies a stemming process to reduce $received$ to its root form: $receive$. The $TeALGenerator$ also performs a stop word removal process for each argument to filter out the most common, short function words such as *the* and *a*. In

this case, the predicate becomes $receive(\_, message)$. The first argument is blank because Senna does not detect its agent. If Senna processes the text "*A received message ...*," $TeALGenerator$ will consider the $received$ predicate as a fluent $received(\_, message)$.

Senna can identify predicates that are related to verbs. However, a limitation of Senna is that it cannot identify fluents such as $in(system, safeMode)$ from the text "*system is in safe mode*." This is because "is" is not considered as a predicate. In $TeALGenerator$, the fluents such as $in(system, safeMode)$ are identified using the typed dependencies generated by the Stanford Parser. The $TeALGenerator$ identifies the fluent $in(system, safeMode)$ through the typed dependencies $nsubj(is, system)$ and $prep\_in(is, mode)$. The typed dependency $nsubj(is, system)$ shows that $system$ is a subject. The type $prep\_in$ means that "in" is a preposition. These two typed dependencies are connected together because of the word is. Because *system is in a mode* describes a property of the system, $TeALGenerator$ generates a new predicate $in(system, mode)$ as a fluent. The fluent is further changed to $in(system, safeMode)$ because the Stanford parse tree (Figure 5.1) shows that "*safe mode*" is a noun phrase.

Identifying Constraints

The $TeALGenerator$ uses a set of tree regular expressions ($Tregex$) patterns [60] and the Stanford parse tree to identify constraints. The $Tregex$ is a utility that matches regular expression patterns in trees. Stanford Parser provides a method for matching a $Tregex$ pattern to text. For example, "*do action within x seconds after*" is a frequently used temporal constraint in requirements. This constraint can be represented as a $Tregex$ pattern:

$$(PP < ((IN < within) \dots (CD\$ + NNS)))$$

The symbol $A < B$ means that A is immediately dominated by B, and $A\$ + B$ means that A is the immediate left sibling of B. The pattern is composed of a number of tags [65]: PP (prepositional phrase), IN (preposition), CD (cardinal number), and NNS (noun, plural). This regular expression can be matched to a prepositional phrase which starts with

a preposition *within* and ends with a number and noun sequence, such as *10 seconds*. Thus, $(PP < ((IN < within) \ldots (CD\$ + NNS)))$ can be used to extract the phrase "*within 10 seconds.*" Similarly, "*do action in last x time units*" can be represented as the $Tregex$ pattern:

$$((IN < in)..((JJ < last)\$ + (CD\$ + NNS)))$$

where JJ is the tag for "adjective." The symbol $A..B$ means that A precedes B.

This method is based on the fact that there are linguistic patterns frequently used in temporal requirements. I collected temporal constraints from different datasets, summarized a set of temporal patterns, and created their regular expressions. For each pattern, there is a *translation string* that represents the way to translate it into $TeAL$. For instance, I have

$$\textbf{noEarlierThan } \$CD \ \$NNS \textbf{ before}$$

for the "*in last x time units*" pattern. The translation string is discussed below.

There are also patterns that represent non-temporal constraints. The patterns

$$(SBAR < ((IN < if)\$ + S))$$

and

$$(SBAR < ((WHADVP < (WRB < when))\$ + S))$$

$$(SBAR < ((WHADVP < (WRB < once))\$ + S))$$

are used for matching texts of the form "*if something*" or "*when something.*" These texts are commonly used for representing preconditions.

I collected twenty linguistic patterns from real software projects. Thirteen patterns can be used to identify temporal constraints, and seven patterns are for non-temporal constraints. I created $TeAL$ statements for all requirements that contain these linguistic patterns, and concluded the translation strings based on these $TeAL$ statements. Table 5.3 shows all patterns and their translation strings, and explains the meaning of these patterns.

Table 5.3: Constraint Patterns

Pattern	Translation String	Explanation
$(PP < ((IN < within)..$ $(CD\$ + NNS)))$	**noLaterThan** $\$CD$ $\$NNS$ **after**	within the next x time units
$(PP < ((IN < after)\$ + S))$	**after** $\$S$	after some event happened
$((IN < in)..((JJ < last)\$+$ $(CD\$ + NNS)))$	**laterThan** $\$CD$ $\$NNS$ **before**	some event has happened in last x time units
$(SBAR < ((IN < before)\$ + S))$	**before** $\$S$	before some event happens
$FRAG < (ADVP < (NP <$ $(CD\$ + NNS) + (RB < prior))$ $\$ + PP < (to < to))$	**at** $\$CD$ $\$NNS$ **before**	x time units prior to
$((NP < (CD\$ + NNS))\$+$ $(PP < (IN < before)\$ + (S)))$	**at** $\$CD$ $\$NNS$ **before**	x time units before
$(VP < (NP < (NN < interval))$ $\$ + ((IN < of) + (NP$ $< (CD\$ + NNS))))$	**at** $\$CD$ $\$NNS$ **after previous**$\$VP$	at an interval of x time units
$(PP < ((IN < within)\$+$ $(NP < ((NP < (CD\$ + NNS)\$+$ $(PP < (IN < of))))))$	**noLaterThan** $\$CD$ $\$NNS$ **before**	within x time units of
$((NP < (CD\$ + NNS))\$+$ $(PP < (IN < after)\$ + (S)))$	**at** $\$CD$ $\$NNS$ **after**	x time units after
$(VP < (NP < QP < ((DT <$ $every)\$ + (CD\$ + NNS))))$	**at** $\$CD$ $\$NNS$ **after previous** $\$VP$	once every x time units
$(NP < (((QP < ((IN < at)\$+$ $(JJS < least)\$ + CD))\$ + NNS)$ $\$ + (PP < ((IN < after)\$ + S))))$	**laterThan** $\$CD$ $\$NNS$ **after**	at least x time units after

Table 5.3, continued

Pattern	Translation String	Explanation
$(NP < (((QP < ((IN < at)\$+$ $(JJS < least)\$ + CD))\$ + NNS)$ $\$ + (PP < ((IN < before)\$ + S))))$	$\mathbf{laterThan}\ \$CD$ $\$NNS\ \mathbf{before}$	at least x time units before
$((IN < in)..((JJ < first)$ $\$ + (CD\$ + NNS)))$	$\mathbf{noLaterThan}\ \$CD$ $\$NNS\ \mathbf{after\ startTime}$	in the first x time units
$(SBAR < ((IN < if)\$ + S))$	$\mathbf{if}\ \ \$S\ \mathbf{then}$	if some event happens
$(SBAR < ((WHADVP <$ $(WRB < when))\$ + S))$	$\mathbf{if}\ \ \$S\ \mathbf{then}$	when some event happens
$(SBAR < ((WHADVP <$ $(WRB < once))\$ + S))$	$\mathbf{if}\ \ \$S\ \mathbf{then}$	once some event happens
$(VP < (VBZ < makes)\$+$ $(NP < S))$	$\mathbf{causes}\ \ \$S$	something makes some event happen
$(VP < (VBZ < causes)\$+$ $(NP < S))$	$\mathbf{causes}\ \ \$S$	something causes some effect
$(VP < (SBAR < ((IN <$ $unless)\$ + S)))$	$\mathbf{impossible\ \ not}\ \$VP$ $\mathbf{if\ \ not}\ \$S$	cannot do something unless
$(S < (VP < ((VBZ < prevents)$ $\$ + NP)))$	$\mathbf{impossible}\ \ \$NP$ $\mathbf{if}\ \ \$VP$	something prevents some event

$TGTree$

Based on the vocabulary and constraints identified using Senna and the Stanford Parser, $TeALGenerator$ creates a tree, $TGTree$, to represent the system information and how each piece of information is related to the others. Vocabulary and constraints are modeled as two types of $TGTree$ nodes: $predicate$ nodes and $constraint$ nodes. The $TGTree$ is

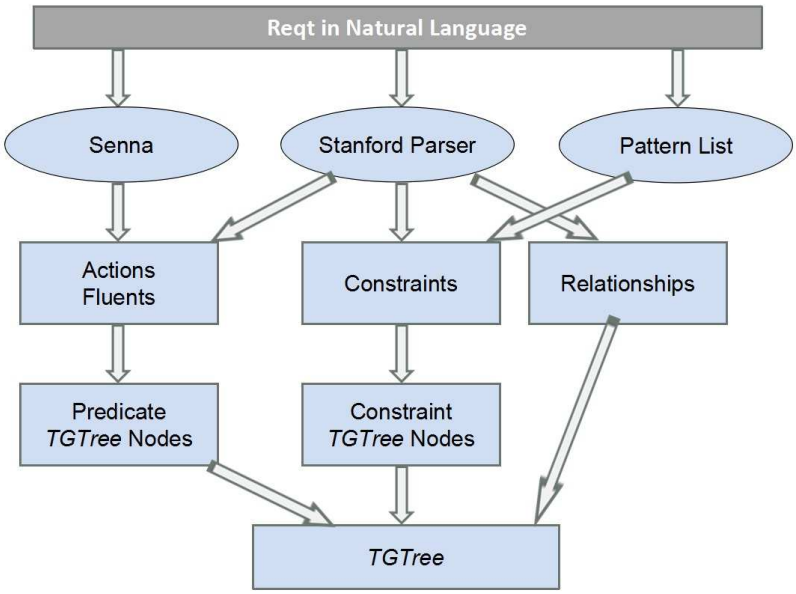


Figure 5.2: Generation of $TGTree$

created based on both types of $TGTree$ nodes and the relationships among these nodes (Figure 5.2). Figure 5.3 shows a $TGTree$ that represents the text "*If the system is not in safe mode or a message is received in last 10 second.*" This $TGTree$ consists of five nodes, where the node 3 ($received(\_, message)$) is identified by Senna, and the other four pieces of information are identified using Stanford Parser.

$TGTree$ **Node.** Vocabulary and constraints are modeled as two types of $TGTree$ nodes: $predicate$ nodes and $constraint$ nodes. In the $TGTree$ in Figure 5.3, node 3 ($received(\ldots)$) and node 4 ($in(system, safeMode)$) are of type $predicate$. These two nodes store the vocabulary information. Node 1 (**if** ...) and node 5 (**noEarlierThan** ... **before**) are of type $constraint$. Node 1 represents a non-temporal constraint, and node 5 represents a temporal constraint. Node 2 (**or**) shows that the relationship between node 3 and node 4 is disjunction.

The list of $TGTree$ node attributes is given below:

– predicate: points to the predicate that is mapped to this node. The value can be $NULL$.

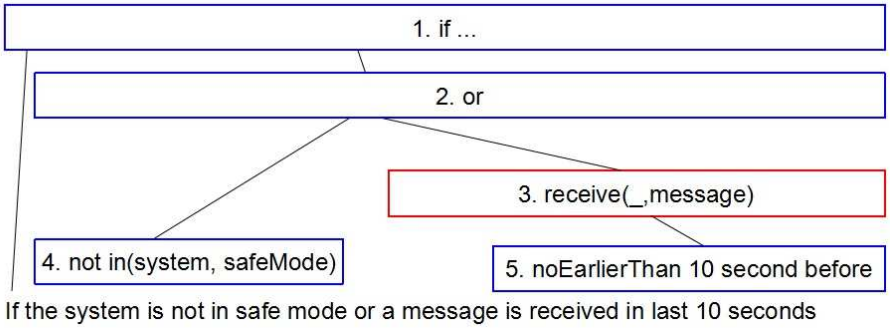


Figure 5.3: Example of $TGTree$

– node: points to the linked node in the Stanford parse tree.

– isNegation: tells if the context represented by this node contains negation.

– isDisjunction: tells if there is disjunction among the children of this node.

– isConjunction: tells if there is conjunction among the children of this node.

– isNonTempRelation: tells if the context represented by this node is a precondition.

– isTempRelation: tells if the context represented by this node is a temporal condition.

– isAction: tells if the predicate mapped to this node (if exist) is an action.

– isFluent: tells if the predicate mapped to this node (if exist) is a fluent.

– content: tells the context represented by this node.

– startPosition: gives the position of the first character of the covered phrase in the text.

– endPosition: gives the position of the last character of the covered phrase in the text.

A $TGTree$ node is a $predicate$ node if its $predicate$ attribute is not $NULL$. The boolean attributes $isAction$ and $isFluent$ illustrate if a $predicate$ node represents an action or a fluent. Similarly, the boolean attributes $isTempRelation$ and $isNonTempRelation$ show if a $constraint$ node represents a temporal or non-temporal constraint. If a $TGTree$ does

not store vocabulary or constraint information, then it is a node of $relation$ type. The $relation$ nodes represent the relation among the other two types. The boolean attributes $isDisjunction$ and $isConjunction$ show if the relation is conjunction or disjunction. The $isNegation$ attribute is true if the corresponding text contains negation. For example, the $TGTree$ node that represents the node 4 ($\mathbf{not}\ in(system, safeMode)$) in Figure 5.3 has this attribute valued as true. A $TGTree$ node also has attributes that are common for all tree nodes, such as $isRoot$, $isLeaf$, $parent$, and $children$.

As is shown in the $TGTree$ (Figure 5.3), each $TGTree$ node has its $content$. For example, the $content$ of node 5 is $\mathbf{noEarlierThan}\ 10\ second\ \mathbf{before}$, and the $content$ of node 3 is $received(\_, message)$. This attribute is used during the construction of $AlmostTeAL$ statements. The evaluation of $content$ is discussed below.

The $node$ attribute points to a node in the parse tree generated by the Stanford Parser. For example, each $TGTree$ node in Figure 5.3 is linked to a parse tree node in Figure 5.1. For each linked pair of nodes, the phrase represented by the parse tree node is "*covered*" by the linked $TGTree$ node. Table 5.4 shows the linked parse tree node for each $TGTree$ node in the $TGTree$ (Figure 5.3): The structures of $TGTree$ trees are determined by the

Table 5.4: Linked $TGTree$ and Parse Tree Nodes

$TGTree$ Node	Parse Tree Node	Covered Phrase
1	1. $SBAR$	If the system is not in safe mode or a message is received in last 10 seconds
2	2. S	the system is not in safe mode or a message is received in last 10 seconds
3	8. S	a message is received in last 10 seconds
4	3. S	the system is not in safe mode
5	12. PP	in last 10 seconds

"*covered phrases*." Given two $TGTree$ nodes n_1, n_2 and their *covered phrases* c_1, c_2, if c_1 is a substring of c_2, then n_1 is a child (or grandchild) of n_2. In the example in Table 5.4, node 3 is the parent of node 5, but node 4 and node 3 are siblings.

The $TeALGenerator$ saves all $TGTree$ nodes in a list, $cList$.

$TGTree$ **Predicate Node.** The $TeALGenerator$ creates a $TGTree$ node of $predicate$ type for each predicate identified by Senna. The algorithm for *generating predicate nodes based on Senna* is given as Algorithm 2. This algorithm uses the Senna output $sennaInfo$ and the parse tree generated by Stanford parser. The node 3 ($received(\ldots)$) in Figure 5.3 is identified using this algorithm.

Algorithm 2 Generating predicate nodes based on Senna

Data: sennaInfo, parseTree, cList

Result: cList

foreach *predicate p* ***in*** *sennaInfo* **do**
    create TGTree node c;
    get the node n in parseTreethat links to c;
    c.node = n;
    c.predicate = p;
    c.startPosition = the start position of n;
    c.endPosition = the end position of n;
    if *the POS tag of p.head is VB VBZ, or VBG* **then**
        c.isAction = true;
    end
    else
        if *the chunk tag of p.head is VP* **then**
            c.isAction = true;
        end
        else
            c.isFluent = true;
        end
    end
    add c to cList;
end

The algorithm for *generating predicate nodes based on Stanford parser* is given as Algorithm 3. This algorithm uses the parse tree and the typed dependencies generated by the Stanford parser. The node 4 ($in(system, safeMode)$) in Figure 5.3 is identified using this algorithm.

Algorithm 3 Generating predicate nodes based on Stanford parser

Data: parseTree, typeDependency, cList

Result: cList

foreach *typed dependency td* ***in*** *typeDependency* **do**

  if *td.reln == prep_x* **then**

    foreach *typed dependency td2* ***in*** *typeDependency* **do**

      if *td2.dep == td.gov* **then**

        create new TGTree node c;

        get the node n in parseTree that links to td;

        c.node = n;

        c.predicate = td.gov;

        c.startPosition = the start position of n in parseTree;

        c.endPosition = the end position of n in parseTree;

        c.isFluent = true;

        add c to cList;

      end

    end

  end

end

The generated $TGTree$ node links to a Stanford parse tree node that is the lowest common ancestor of all the arguments of the predicate. In Figure 5.1, the node "8. S" is the lowest ancestor of *a message* and *received in last 10 seconds*. Thus, the $TGTree$ node 3 in Figure 5.3 links to "8. S" and covers the text "*a message is received in last 10 seconds*."

Similarly, the node "3. S" in Figure 5.1 is the lowest ancestor of *system* and *mode*. In this case the $TGTree$ node 4 in Figure 5.3 links to "3. S" and covers the text "*the system is not in safe mode*."

$TGTree$ **Constraint Node.** The $TeALGenerator$ tries to find matching phrases for all the patterns I created. For each matching phrase, $TeALGenerator$ creates a $constraint$ type $TGTree$ node. The algorithm for *generating constraint nodes* is given as Algorithm 4. This algorithm uses the Stanford parse tree and the pattern list I created.

Algorithm 4 Generating constraint nodes

Data: parseTree, patternList, cList

Result: cList

foreach *temporal pattern r* ***in*** *patternList* **do**

    if *r has a match rp in parseTree* **then**
        create new TGTree node c;
        if *r is a temporal pattern* **then**
            c.isTempRelation = true;
        end
        else
            c.isNonTempRelation = true;
        end
        c.node = rp.root;
        c.startPosition = the start position of rp.root;
        c.endPosition = the end position of rp.root;
        c.content = the translation string of r;
        add c to cList;
    end

end

A $constraint$ type $TGTree$ node is linked to the matching parse tree node. For example,

the $TGTree$ for "*in last 10 seconds*" (node 5 in Figure 5.3) is linked to the node "12. PP" in Figure 5.1. The $content$ of a $constraint$ type $TGTree$ node is set as the corresponding *translation string*, and the tags marked with $ sign mean that these tags will be replaced with the corresponding text in the requirements when $AlmostTeAL$ is generated.

$TGTree$ **Construction.** The $TeALGenerator$ builds a $TGTree$ based on all the $predicate$ and $constraint$ $TGTree$ nodes. The root of the $TGTree$ covers the whole sentence. Then the $TGTree$ nodes are constructed based on the texts they cover. The text that is covered by a child node must be a substring of the text covered by the parent node. In the example in Figure 5.3, node 5 is the child of node 3 because of the texts they cover.

The algorithm for *constructing a* $TGTree$ is given as Algorithm 5. It uses the $cList$ that includes all *predicate* and *constraint* $TGTree$ nodes.

Algorithm 5 Constructing a $TGTree$

Data: cList

Result: TGTree

foreach *TGTree node n1* ***in*** *cList* **do**
    find the node n2 such that n2 covers n1
    and there is no n3 such that n2 covers n3 and n3 covers n1;
    add n1 to n2.children;
    n1.parent = n2;
    if *n1.parent = null* **then**
        n1.isRoot = true;
    end
    if *n1.children = null* **then**
        n1.isLeaf = true;
    end
end

After the $TGTree$ is built, the $TeALGenerator$ creates $TGTree$ nodes of $relation$ type

based on Stanford Dependencies. The $TeALGenerator$ identifies three types of relations among constraints: negation, conjunction, and disjunction.

The algorithm for *identifying conjunction, disjunction* is given as Algorithm 6. This algorithm uses the type dependencies generated by Stanford Parser and the $TGTree$ generated in Algorithm 5.

Algorithm 6 Identifying conjunction, disjunction

Data: typeDependency, TGTree

Result: TGTree

foreach *typed dependency td* ***in*** *typeDependency* **do**

    if *td.reln == conj_or or conj_and* **then**
        get the lowest TGTree node that covers td.gov, n1;
        get the lowest TGTree node that covers td.dep, n2;
        get the lowest common ancestor, n3;
        create new tempCond node n4;
        set n4 to be the new parent of n1 and n2;
        add n4 to n3.children;
        if *td.reln == conj_and* **then**
            n4.isConjunction = true;
        end
        else
            n4.isDisjunction = true;
        end
    end

end

For disjunction and conjunction relationships, the useful dependencies are $conj\_or$ and $conj\_and$. Given the sample text and the dependency $conj\_or(is, received)$ (No. 11 in Table 5.2), $TeALGenerator$ creates a new $TGTree$ node which is of type "$Disjunction$." The two $TGTree$ nodes that represent the predicates $receive$ and in become the children

of this new $TGTree$ node. The text of the new node covers the texts of both its children. The new node is then inserted into the $TGTree$ based on the text it covers. By default, I assume that the relationship between the children $TGTree$ nodes under the same parent is conjunction, because many sentences do not present the conjunction relation explicitly.

The negation relation is identified through the dependency $neg(x, not)$. Given such a dependency, the $TeALGenerator$ checks if there is a $TGTree$ node that matches the word x. In the "*system is in mode*" example above, if there is a dependency $neg(is, not)$, then the $TGTree$ node that corresponds to this phrase is marked as negation.

Generating $AlmostTeAL$

The $TeALGenerator$ constructs the signature Σ of $AlmostTeAL$ based on the $predicate$ $TGTree$ nodes. In the example "*If the system is not in safe mode or a message is received in last 10 seconds*" two $predicate$ $TGTree$ nodes are created: action node $receive(\_, message)$ and fluent node $in(system, safeMode)$. The $TeALGenerator$ identifies three sorts from the arguments of these nodes, and each sort has one constant:

$$\textbf{sort } systemSort,\ safeModeSort,\ messageSort;$$

$$\textbf{constant } systemSort\ system;$$

$$\textbf{constant } safeModeSort\ safeMode;$$

$$\textbf{constant } messageSort\ message;$$

The declaration of the action is:

$$\textbf{action } receive(\_, messageSort);$$

The declaration of the fluent is:

$$\textbf{fluent } in(systemSort, safeModeSort);$$

The $TeALGenerator$ constructs $AlmostTeAL$ statements based on the structure of the $TGTree$. The process of creating $AlmostTeAL$ statements is based on a depth-first search.

For a $TGTree$ node of constraint type, its $content$ attribute is the *translation string* of the pattern that matches this node. If the $content$ attribute contains tags that are marked with $ sign, then these tags must be replaced with the corresponding text in the requirements. For example, given a constraint $TGTree$ node for the *in last 10 seconds* text, its $content$ attribute is

$$\textbf{noEarlierThan}\ \$CD\ \$NNS\ \textbf{before}$$

Thus $TeALGenerator$ will output $\textbf{noEarlierThan}\ 10\ second\ \textbf{before}$ because the tags $CD\ NNS$ match "*10 seconds*." For a $TGTree$ node of $predicate$ type, its $content$ attribute is of the form

$$predicateName(Arg_1, \ldots, Arg_n)$$

$TeALGenerator$ outputs the $content$ attribute without any change.

From the tree given in Figure 5.3, the $TeALGenerator$ generates the $AlmostTeAL$ expression:

$$\textbf{if not}\ in(system, safeMode)\ \textbf{or}$$
$$receive(\_, message)\ \textbf{noEarlierThan}\ 10\ second\ \textbf{before}$$

5.2 Detecting Ambiguity and Incompleteness

During the process of identifying system information, $TeALGenerator$ may also detect ambiguity and incompleteness and use this information to assist analysts' work.

The approach to detect ambiguity is similar to that of identifying constraints. I identified a set of "*ambiguity patterns*" based on existing ambiguity detection research [11] and my own experience. For instance, given a pattern "*do action every x seconds*," it is not clear if the *action* is performed at an interval of exactly *x seconds*, or at most *x seconds*. Another example is the "*in last 10 seconds*" phrase above. The $TeALGenerator$ assumes that this phrase means $\textbf{noEarlierThan}\ 10\ second\ \textbf{before}$. But some analysts may consider this

phrase as $\mathbf{laterThan}\ 10\ second\ \mathbf{before}$. This problem must be clarified by the analysts before a correct $TeAL$ theory is generated.

Detecting incompleteness is based on the assumptions of $TeAL$, such that each action must have effect on the system, and that each action must have an agent. If Senna finds a predicate without any agent (for instance, $receive(\_, message)$), then $TeALGenerator$ will remind analysts that an agent is missing. Another case of incompleteness is that $TeALGenerator$ identifies a fluent such as $in(system, safeMode)$, but it does not appear in the "*effect*" part (after $\mathbf{then}$) in any $AlmostTeAL$ statements. The $TeALGenerator$ will remind analysts that this fluent is not affected by any action.

The $TeALGenerator$ also checks if there are missing arguments by comparing different predicates. These predicates may be identified from different requirements. For instance, $TeALGenerator$ uses Stanford Dependencies to decide the types of some arguments. With the typed dependency $prep\_from(receiver, sender)$, the tool decides that the *receive* action has an argument whose type is "*receive from.*" This information is useful because it is very possible that a requirement does not contain information about all the arguments. For example, given the predicate $receive(\_, message)$, Senna does not consider that the action has an argument representing where the message comes from. With the argument "*receive_from*" identified from the other requirements, the action predicate $receive(\_, message)$ can be changed to $receive(\_, message, \_)$. This reminds analysts that a third argument representing "*receive from somewhere*" is missing.

Chapter 6 From $TeAL$ to $clingcon$ language

Once analysts confirm that a $TeAL$ theory is correct, their task in the process of generating $TeAL$ theory is complete. The next step is to generate an answer set program based on the $TeAL$ theory. An answer set program is a logic-based formalism which maps search problems to logic expressions and offers tools to reason with these logic expressions automatically. I designed $TeAL$ as an extension of the action language AL [9], and there is the translation from AL to ASP.

This step is fully automated by my translator $TeAL2ASP$. The translation is designed so that there is a correspondence between a valid timed path of a $TeAL$ theory and an answer set of the answer set program translated from that $TeAL$ theory.

I chose $clingcon$ [35] as the tool for processing the answer set programs translated from $TeAL$ theories. The $clingcon$ programs combine ASP programs with constraints over integers. The use of these constraints makes $clingcon$ more efficient in handling large numeric domains.

As I have discussed in Chapter 4, given a $TeAL$ theory $\Delta = \langle \Sigma, AD, TC \rangle$, I extend it to a normalized theory $\Delta^N = \langle \Sigma^N, AD^N, TC \rangle$. I write $\Sigma(\Delta^N)$ to denote the *signature* of Δ^N, $AD(\Delta^N)$ to denote the *action description* of Δ^N, and $TC(\Delta)$ to denote the set of *temporal constraints*. The translation of Δ^N, $\Pi(\Delta^N)$, is a $clingcon$ program that consists of three parts: $\Pi(\Sigma(\Delta^N))$, $\Pi(AD(\Delta^N))$, and $\Pi(TC(\Delta))$, expressing the three components of Δ^N, respectively. The $clingcon$ program $\Pi(AD(\Delta^N))$ involves a parameter, say NS, that represents the number of states on paths. Answer sets of $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N))$ represent timed paths with $NS + 1$ states (the initial state and the NS states after it) in the transition graph of Δ^N. I write $\Pi(AD(\Delta^N), n)$ to represent the program $\Pi(AD(\Delta^N))$ with NS set to n. The program $\Pi(TC(\Delta))$ involves another parameter, say H, that represents the horizon of timed paths. I write $\Pi(TC(\Delta), h)$ to denote the program $\Pi(TC(\Delta))$

with the horizon value set to h. I will discuss the generation of the programs $\Pi(\Sigma(\Delta^N))$, $\Pi(AD(\Delta^N), n)$, and $\Pi(TC(\Delta), h)$ below.

6.1 Generation of $\Sigma(\Delta^N)$

The program $\Pi(\Sigma(\Delta^N))$ is the translation of $\Sigma(\Delta^N)$, the normalized signature of the $TeAL$ theory Δ. The program $\Pi(\Sigma(\Delta^N))$ defines sorts, their extensions, fluents, actions, and agents. These elements are translated into atoms and rules in $\Pi(\Sigma(\Delta^N))$.

For every constant declaration "**constant** s $con_1, \ldots, con_k$;" in $\Sigma(\Delta^N)$, where s must be a sort declared in $\Sigma(\Delta^N)$, I include in $\Pi(\Sigma(\Delta^N))$ the statements:

$$s(con_1). \quad \ldots \quad s(con_k). \tag{6.1}$$

In $clingcon$, these statements define the extension of the predicate s, and indicate that constants $con_1, \ldots, con_k$ are of sort s.

For every agent declaration "**agent** $ag_1, \ldots, ag_k$;" in $\Sigma(\Delta^N)$, where $ag_1, \ldots, ag_k$ are sorts that are declared as agents, I include in $\Pi(\Sigma(\Delta^N))$ the statements:

$$\begin{gathered} agent(Ag) \text{:- } ag_1(Ag). \\ \ldots \\ agent(Ag) \text{:- } ag_k(Ag). \end{gathered} \tag{6.2}$$

These statements define the extension of the predicate $agent$, and indicate that all constants of the sorts $ag_1, \ldots, ag_k$ are agents.

For each fluent declaration "**fluent** $fluentName(s_1, \ldots, s_k)$;" in $\Sigma(\Delta^N)$, I include in $\Pi(\Sigma(\Delta^N))$ the statement:

$$fluent(fluentName(C_1, \ldots, C_k)) \text{:- } s_1(C_1), \ \ldots \ , \ s_k(C_k). \tag{6.3}$$

This defines the space (set) of fluents. I use the $fluent$ predicate to indicate that a valid fluent $fluentName(\ldots)$ must include constants $C_1, \ldots, C_k$ that are of correct sorts $s_1 \ldots, s_k$.

In normalized $TeAL$ theories prompts play the role of actions in AL. For each prompt "**commence** $actionName(s_1, \ldots, s_k)$" or "**terminate** $actionName(s_1, \ldots, s_k)$" in the signature $\Sigma(\Delta^N)$, I include in $\Pi(\Sigma(\Delta^N))$ the statements:

$$act(actionName(C_2, \ldots, C_k)) \text{:- } s_2(C_2),\ \ldots\ ,\ s_k(C_k). \tag{6.4}$$

$$action(Ag, actionName(C_2, \ldots, C_k)) \text{:- } s_1(Ag), s_2(C_2),\ \ldots\ ,\ s_k(C_k). \tag{6.5}$$

$$prompt(com(action(Ag, Ac))) \text{:- } action(Ag, Ac). \tag{6.6}$$

$$prompt(ter(action(Ag, Ac))) \text{:- } action(Ag, Ac). \tag{6.7}$$

These rules define the space (set) of prompts. Because $TeAL$ assumes that each action must have an agent, I use the act predicate to represent the actions in AC without specifying their agents. I use the second rule to specify the agents (Ag) of these actions. For instance, $act(send(message, receiver))$ stands for "*sending message to the receiver*," and $action(sender, send(message, receiver))$ specifies that it is $sender$ that performs this action. The third and forth rules declare the **commence** and **terminate** prompts, where $com(action(Ag, Ac))$ represents the **commence** $action(Ag, Ac)$ prompt, and $ter(action(Ag, Ac))$ represents the **terminate** $action(Ag, Ac)$ prompt. Additionally, I include in $\Pi(\Sigma(\Delta^N))$ the statements:

$$prompt(totrue(F)) \text{:- } \mathit{fluent}(F). \tag{6.8}$$

$$prompt(tofalse(F)) \text{:- } \mathit{fluent}(F). \tag{6.9}$$

to define that for every fluent F, $totrue(F)$ and $tofalse(F)$ are prompts.

Introduction of engaged and inProgress. Because Σ^N includes two types of special fluents **engaged** Ag and **inProgress** Act for all agents Ag in AG and actions Act in AC, I include in $\Pi(\Sigma(\Delta^N))$ the statements:

$$\mathit{fluent}(engaged(Ag)) \text{:- } agent(Ag). \tag{6.10}$$

$$\mathit{fluent}(\mathit{progress}(\mathit{action}(Ag, \mathit{actionName}(C_2, \ldots, C_k))))$$
$$\text{:- } ag_1(Ag), s_2(C_2), \ \ldots \ , \ s_k(C_k). \tag{6.11}$$

6.2 Generation of $\Pi(AD(\Delta^N), n)$

Next I describe the translation of the action theory laws. The translation of state constraints, dynamic causal laws, and executability conditions follows Baral and Gelfond's work [9].

Predicate $state$**.** Following Baral and Gelfond, I introduce a new predicate $state$ in $\Pi(AD(\Delta^N), n)$. The program $\Pi(AD(\Delta^N), n)$ uses the following statement to identify the $state$ predicate with integers from 0 to n.

$$state(0..n). \tag{6.12}$$

Introduction of $holds$ **and** $happen$**.** As in Baral and Gelfond's work, I include two predicates, $holds$ and $happen$ in the $clingcon$ program to represent the relationships among fluents, prompts, and states. The $\Pi(AD(\Delta^N), n)$ program uses the predicate:

$$holds(F, S) \tag{6.13}$$

to represent that the value of fluent F is $true$ at state S. The $\Pi(AD(\Delta^N), n)$ program also uses the predicate:

$$happen(Pr, S) \tag{6.14}$$

to represent that prompt Pr happens at state S, and changes the system to the next state.

Translation of *action description* $AD(\Delta^N)$**.** For every state constraint

$$L \textbf{ if } P;$$

I include in $\Pi(AD(\Delta^N), n)$ the following rule:

$$holds(L, S)\text{:- } holds(P, S), state(S), \mathit{fluent}(P), \mathit{fluent}(L). \tag{6.15}$$

If P is the negation of a fluent P', then I replace $holds(P, S)$ with -*holds* (P', S). I applies this method to all negation fluents in Δ^N. If P is a conjunction of fluents $P_1, \ldots, P_k$, then I include in $\Pi(AD(\Delta^N), n)$ the following rule:

$$holds(L, S) \text{:- } holds(P_1, S), \mathit{fluent}(P_1), \ldots, holds(P_k, S), \mathit{fluent}(P_k), \\ state(S), \mathit{fluent}(L). \tag{6.16}$$

If P is a disjunction of fluents $P_1, \ldots, P_k$, then I include in $\Pi(AD(\Delta^N), n)$ the following rule:

$$holds(L, S) \text{:- } 1\{holds(P_1, S), \ldots, holds(P_k, S)\}, state(S), \\ \mathit{fluent}(P_1), \ldots, \mathit{fluent}(P_k), \mathit{fluent}(L). \tag{6.17}$$

If L is a conjunction of fluents $L_1, \ldots, L_k$, then I include in $\Pi(AD(\Delta^N), n)$ the following rule:

$$k\{holds(L_1, S), \ldots, holds(L_k, S)\} \text{:-} \\ holds(P, S), \mathit{fluent}(L_1), \ldots, \mathit{fluent}(L_k), \\ state(S), \mathit{fluent}(P). \tag{6.18}$$

If L is a disjunction of fluents $L_1, \ldots, L_k$, then I include in $\Pi(AD(\Delta^N), n)$ the following rule:

$$1\{holds(L_1, S), \ldots, holds(L_k, S)\} \text{:-} \\ holds(P, S), \mathit{fluent}(L_1), \ldots, \mathit{fluent}(L_k), \\ state(S), \mathit{fluent}(P). \tag{6.19}$$

For every dynamic causal law

$$Pr \textbf{ causes } E \textbf{ if } P;$$

I could include in $\Pi(AD(\Delta^N))$ the following rule:

$$holds(E, S+1) \text{:- } happen(Pr, S), holds(P, S), state(S)$$
$$prompt(Pr), \mathit{fluent}(P), \mathit{fluent}(E), S <= n-1. \tag{6.20}$$

This is the standard way to translate dynamic laws to logic program rules. However, it should be noted that a fluent may change because of two reasons: the occurrence of a regular prompt, and passing time. While the rules above cover the first case, we can use virtual prompts $totrue(\ldots)$ and $tofalse(\ldots)$ to cover both cases by assuming that whenever a regular prompt that changes the value of a fluent occurs, the corresponding virtual prompt must occur as well. This assumption can be enforced by the following rules:

$$\text{:- } happen(Pr, S),\ not\ happen(totrue(F), S),$$
$$state(S), prompt(Pr), prompt(totrue(F)). \tag{6.21}$$
$$\text{:- } happen(Pr, S),\ not\ happen(tofalse(F), S),$$
$$state(S), prompt(Pr), prompt(tofalse(F)). \tag{6.22}$$

The following rules describe how $totrue(\ldots)$ and $tofalse(\ldots)$ change fluents.

$$holds(F, S+1) \text{:- } happen(totrue(F), S), \mathit{fluent}(F), state(S),$$
$$S <= n-1. \tag{6.23}$$
$$\textit{-holds}\ (F, S+1) \text{:- } happen(tofalse(F), S), \mathit{fluent}(F), state(S),$$
$$S <= n-1. \tag{6.24}$$

Thus the following rules are sufficient for capturing all dynamic causal laws

$$Pr \textbf{ causes } E \textbf{ if } P;$$

where E can be a fluent F or its negation. If E represents a fluent F, then the rules are:

$$happen(totrue(F), S) \text{:- } happen(Pr, S), holds(P, S), state(S),$$
$$prompt(Pr), \mathit{fluent}(P), \mathit{fluent}(F). \tag{6.25}$$

If E represents the negation of a fluent F, then the rules are:

$$happen(tofalse(F), S) \text{:- } happen(Pr, S), holds(P, S), state(S), \\ prompt(Pr), fluent(P), fluent(F). \quad (6.26)$$

For every executability condition

$$\textbf{impossible } pr_1, \ldots, pr_k \textbf{ if } P;$$

I include in $\Pi(AD(\Delta^N))$ the following rule:

$$\text{:- } happen(Pr_1, S), \ldots, happen(Pr_k, S), holds(P, S), \\ state(S), fluent(P), prompt(Pr_1), \ldots, prompt(Pr_k). \quad (6.27)$$

Because of the existence of $totrue()$ and $tofalse()$ prompts and the assumption above, I use rules that are slightly different from Baral and Gelfond's work [9] to represent the change of fluents and the inertial axioms.

$$holds(F, S+1) \text{:- } holds(F, S), not\ happen(tofalse(F), S), \\ fluent(F), state(S). \quad (6.28)$$

$$\textit{-holds}\ (F, S+1) \text{:- } \textit{-holds}\ (F, S), not\ happen(totrue(F), S), \\ fluent(F), state(S). \quad (6.29)$$

$$\text{:- } not\ holds(F, S), not\ \textit{-holds}\ (F, S), fluent(F), state(S). \quad (6.30)$$

The rules 6.28 and 6.29 specify that a fluent will not change unless something (prompts or passing time) makes it change. I use $totrue()$ and $tofalse()$ because we ensure that no matter whether a fluent F is inertial or not, $totrue(F)$ must happen at state S in order for $holds(F, S+1)$ to become $true$, and $tofalse(F)$ must happen at state S in order for $holds(F, S+1)$ to become $false$. The rule 6.30 means the value of a fluent must be either $true$ or $false$. It ensures that states are complete.

The following rule guarantees that at each state at least one prompt occurs, because only prompts can change the states of the system.

$$1\{happen(Pr, S) : prompt(Pr)\}\text{:- } state(S). \tag{6.31}$$

The constraint above is typically implicit in textual requirement documents and in $TeAL$. However, it must be made explicit in the $clingcon$ program.

For every initial constraint **initially** F, $\Pi(AD(\Delta^N), n)$ includes the rule:

$$holds(F, 0)\text{:- } init(F). \tag{6.32}$$

If F is the negation of a fluent F', then $\Pi(AD(\Delta^N), n)$ includes the rule:

$$\textit{-holds } (F', 0)\text{:- } init(F'). \tag{6.33}$$

Given an integer n and a normalized $TeAL$ theory Δ^N, $\Sigma(\Delta^N) \cup AD(\Delta^N)$ is an AL theory. Because the translation of $\Sigma(\Delta^N) \cup AD(\Delta^N)$ is essentially that of Baral and Gelfond [9], the following theorem is a restatement of their result (Theorem 2, page 11 [9]).

Theorem 3 *If a sequence $p = \langle s_0, pr_0, \ldots, s_{n-1}, pr_{n-1}, s_n\rangle$ is a valid path in T_{Δ^N}, then $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$ has an answer set A such that for every i, $0 \leq i \leq n$,*

1. *for every fluent f, $holds(f, i) \in A$ if and only if $f \in s_i$*
2. *for every prompt pr, $happen(pr, i) \in A$ if and only if $pr \in pr_i$*

Conversely, if A is an answer set of $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$, then there is a valid path $p = \langle s_0, pr_0, \ldots, s_{n-1}, pr_{n-1}, s_n\rangle$ in T_{Δ^N} such that for every i, $0 \leq i \leq n$,

1. *for every fluent f, $f \in s_i$ if and only if $holds(f, i) \in A$*
2. *for every prompt pr, $pr \in pr_i$ if and only if $happen(pr, i) \in A$*

6.3 Generation of $\Pi(TC(\Delta), h)$

As I have discussed in Chapter 4, my program aims to detect inconsistencies within a given time horizon. The value of time horizon must be given in the $clingcon$ program. I write $\Pi(TC(\Delta), h)$ to indicate that in the program $\Pi(TC(\Delta))$ the horizon value is h.

The program $\Pi(TC(\Delta), h)$ uses the following statement to define the time domain and represent that the value of the horizon is h:

$$\$domain(0..h). \tag{6.34}$$

The $\$domain$ defines the possible values of integer variables in this $clingcon$ program. The $ sign means that the time domain is treated separately when $\Pi(TC(\Delta), h)$ is grounded. In the $clingcon$ program described below I use $time(C)$ as a constraint variable to represent the time moment assigned to the check point C. These variables are valued as integers in $[0, h]$. They are not grounded like the other variables. The integer constraints associated with these variables are marked with $ and are solved in $clingcon$ by a dedicated constraint solver $Gecode$ [79]. For example, given a rule

$$time(C1)\$ \leq time(C2). \tag{6.35}$$

the $Gecode$ solver will need to assign integer values $int1$ and $int2$ to variables $time(C1)$ and $time(C2)$ so that the inequality $int1 \leq int2$ holds. The use of $Gecode$ prevents the generation of huge numbers of ground instances of rules. The range of the integers is defined by the statement $\$domain(0..h)$, which means that the values assigned to $time(C1)$ and $time(C2)$ must be less or equal to h and greater than or equal to 0. This ensures that all check points are assigned time moments within the horizon.

As discussed in Chapter 4, we need to check if temporal conditions are satisfied on timed paths, but it is not necessary to check every time moment. For every temporal condition, we can find a finite set of *check points* such that checking the satisfiability of temporal conditions on these check points is equivalent to checking the satisfiability at every time moment on the path (Theorem 1).

As mentioned in Chapter 4, there are two types of check points: state determined check points associated with states, and condition determined check points associated with temporal constraints. The number of state determined check points equals to n+1, where n is the number of states, because the start time is also regarded as a state determined check point. The number of condition determined check points is a priori unknown. Because of this, I include the following rule in $\Pi(TC(\Delta), h)$ to declare the upper bound of the number of candidate check points:

$$check(0..h). \tag{6.36}$$

The $check()$ predicate has a range from 0 to h. This rule indicates that the number of candidate check points equals to the number of time moments on the timed path. My translation introduces a certain number of check points and numbers them consecutively (check point 0, check point 1, etc., up to check point k, for some $k \leq h$). Checking the satisfiability of temporal constraints on these k *sufficient check points* will be enough to determine if the timed path is valid. The enumeration of check points is consistent with their times.

I use a predicate $chp(C)$ to represent a sufficient check point C. Each of these check points has a time assigned to it. The following rule ensures that the time assignment of check points is based on the sequence of the check points on the path. I use $time(C1)$ and $time(C2)$ to represent the times assigned to $C1$ and $C2$.

$$\begin{aligned} \text{:- } & chp(C1), chp(C2), C1 > C2, \\ & time(C1)\$ < horizon, time(C1) \leq time(C2). \end{aligned} \tag{6.37}$$

Figure 6.1 shows an example in which the horizon of the timed path p is 21. The upper bound of the number of candidate check points is set to be 20, but there are only six sufficient check points ($k = 5$). Four state determined check points ($chp(0)$, $chp(1)$, $chp(3)$, $chp(5)$) map the start time and the states $s0$, $s1$, and $s2$. There are also two condition de-

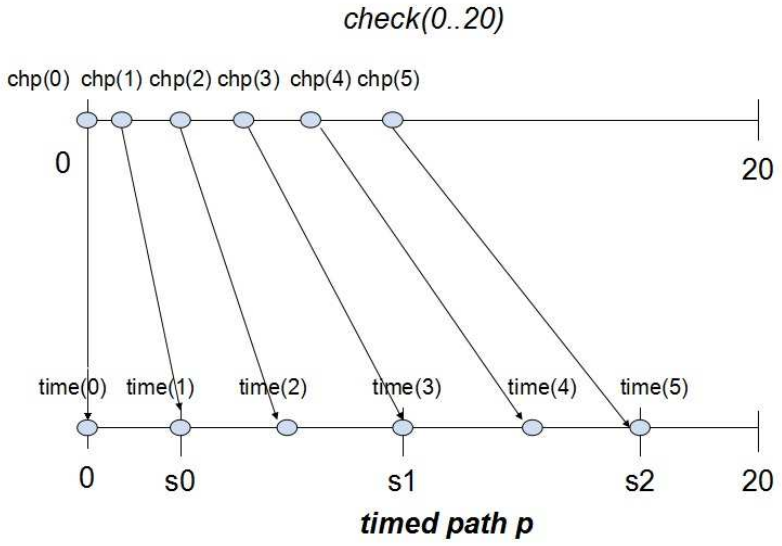


Figure 6.1: Example of check points on timed path

termined check points ($chp(2)$ and $chp(4)$). The order of check points and the order of the times assigned to these check points are consistent.

The state determined check points are the time moments when prompts occur and change the system. These time moments are the last time moments before the states change. In $\Pi(TC(\Delta), h)$ I use a predicate $statechp(C, S)$ to represent that the state S is mapped to a state determined check point C. An atom $statechp(C_i, S_j)$ indicates that if a prompt occurs at state S_j, then this prompt occurs at $time(C_i)$, the last time moment when the system is still in S_j. The rule enforcing that mapping is:

$$1\{statechp(C, S) : check(C)\}1 \text{ :- } state(S). \tag{6.38}$$

It means that a state must be mapped to one and only one state determined check point. The rule below specifies that every state determined check point needs to be checked

$$chp(C) \text{ :- } statechp(C, S). \tag{6.39}$$

In $\Pi(TC(\Delta), h)$ I include two predicates $prompthappen$ and $fluentholds$ as the "check point" version of $happen$ and $holds$. They specify that "a fluent holds at a check point" and

"a prompt happens at a check point." Their definitions are:

$$prompthappen(Pr, C) \text{ :- } happen(Pr, S), statechp(C, S). \tag{6.40}$$

$$\mathit{fluentholds}(F, C) \text{ :- } holds(F, S), statechp(C, S). \tag{6.41}$$

Each temporal condition gives rise to its condition defined check points. In $\Pi(TC(\Delta), h)$ I use atoms $chpAt(ID, CP)$ to represent that "a set of condition determined check points ID is defined by the state determined check point CP."

Let us discuss a temporal conditions "β **at** 2 $unit$ **before** α," where α and β are prompts. The $clingcon$ program needs to ensure that a check point exists *2 units **before** the time when α occurs*. I use $chpAt(cpid1, CP1)$ to represent the existence of such a condition determined check point $cpid1$, where $CP1$ represents the state determined check point when α occurs. We need to specify that this $chpAt()$ atom is associated with $cpid1$ because temporal conditions may use the same set of state determined check points to define different condition determined check points.

I include the following statements in $\Pi(TC(\Delta), h)$ for these check points:

$$\begin{aligned} chp(CP2) \text{:- } & time(CP2)\$ == time(CP1) - 2, check(CP2), \\ & prompthappen(\alpha, CP1). \end{aligned} \tag{6.42}$$

$$\begin{aligned} chpAt(cpid1, CP1) \text{ :- } & check(CP2),\ time(CP2)\$ == time(CP1) - 2, \\ & prompthappen(\alpha, CP1). \end{aligned} \tag{6.43}$$

$$\begin{aligned} \text{:- } & prompthappen(\alpha, CP1),\ not\ chpAt(cpid1, CP1), \\ & 0\$ <= time(CP1) - 2. \end{aligned} \tag{6.44}$$

The first rule specifies that *2 units before the time moment when α happens* is a condition determined check point. The second rule states that $chpAt(cpid1, CP1)$ is $true$ if there is a condition determined check point $CP2$ at 2 units before the state determined check point $CP1$, the time moment when α happens. The third rule means that the condition determined check point $CP2$ must be checked.

For another example, I will discuss the temporal condition "γ **at** 5 $unit$ **after** α." In this case, the $clingcon$ program needs to ensure that a check point exists *at 5 units **after** the time when α occurs*. I include the following statements in $\Pi(TC(\Delta), h)$:

$$chp(CP2) \text{:- } time(CP2)\$ == time(CP1) + 5, check(CP2), \\ prompthappen(\alpha, CP1). \tag{6.45}$$

$$chpAt(cpid2, CP1) \text{ :- } check(CP2),\ time(CP2)\$ == time(CP1) + 5, \\ prompthappen(\alpha, CP1). \tag{6.46}$$

$$\text{:- } prompthappen(\alpha, CP1),\ not\ chpAt(cpid2, CP1), \\ horizon\$ >= time(CP1) + 5. \tag{6.47}$$

to ensure that there is a condition determined check point $CP2$ at 5 units before the state determined check point $CP1$, the time moment when α happens.

Translation of next and previous

The next and previous occurrences of prompts, **next** $prompt$ and **previous** $prompt$, are defined by means of predicates $next$ and $previous$:

$$previous(prompthappen(Pr, S1), S)\text{:-} \\ prompthappen(Pr, S1), statechp(S), S > S1, \\ \{prompthappen(Pr, S2) : statechp(S2) : S2 > S1 : S2 < S\}0. \tag{6.48}$$

$$next(checkhappen(Pr, S1), S)\text{:-} \\ prompthappen(Pr, S1), statechp(S), S1 > S, \\ \{prompthappen(Pr, S2) : statechp(S2) : S1 > S2 : S2 > S\}0. \tag{6.49}$$

The predicate $previous((Pr, S1), S)$ represents that for state S, the previous occurrence of Pr is at state $S1$. Similarly, $next((Pr, S1), S)$ means that for state S, the next occurrence of Pr is at state $S1$.

Translation of Temporal Constraints

As I have discussed in Chapter 4, there are local temporal conditions whose satisfaction depends on time moments, and there are global temporal conditions whose satisfaction does not. The $\Pi(TC(\Delta), h)$ program includes rules for both types.

The $\Pi(TC(\Delta), h)$ program uses $sat(ID, arguments, CP)$ to represent that "*the temporal condition ID is satisfied at the check point CP.*" The translation assigns a unique ID for each temporal condition. The $arguments$ are the actions and fluents involved in the temporal condition. I will now illustrate the translation of temporal conditions with examples.

Based on the discussion in Chapter 4, a temporal condition $C = \alpha\ timeComp\ timeMod\ \beta$ is local if β is empty or β involves **previous** or **next**. Given a local temporal condition $cond1 = \alpha$ **at** c *unit* **after** β_n, where β_n stands for "the next occurrence of β," $\Pi(TC(\Delta), h)$ includes the following rules:

$$sat(cond1, \alpha, \beta, CP1) \text{:- } not\ \textit{-sat}\ (cond1, \alpha, \beta, CP1), chp(CP1). \tag{6.50}$$

$$\begin{aligned}\textit{-sat}\ (cond1, \alpha, \beta, CP1) \text{:- } & next(prompthappen(\beta, CP2), CP1),\\ & not\ prompthappen(\alpha, CP3), chp(CP1),\\ & time(CP3)\$ == time(CP2) + c, chp(CP2),\\ & chp(CP3), horizon\$ >= time(CP2) + c.\end{aligned} \tag{6.51}$$

$$\begin{aligned}\textit{-sat}\ (cond1, \alpha, \beta, CP1) \text{:- } & not\ next(prompthappen(\beta, CP2), CP1),\\ & chp(CP1), chp(CP2).\end{aligned} \tag{6.52}$$

For another example of local temporal condition, let us consider that $cond2$ represents α **at** c *unit* **after**. Then, $\Pi(TC(\Delta), h)$ includes the following rules:

$$sat(cond2, \alpha, CP1) \text{:- } not\ \textit{-sat}\ (cond2, \alpha, CP1), chp(CP1). \tag{6.53}$$

$$\textit{-sat}\ (cond2, \alpha, CP1)\text{:-}\ not\ prompthappen(\alpha, CP2),$$
$$time(CP2)\$ == time(CP1) + c,$$
$$chp(CP1), chp(CP2),$$
$$horizon\$ >= time(CP1) + c. \tag{6.54}$$

Based on the discussion in Chapter 4, a temporal condition $C = \alpha\ timeComp\ timeMod\ \beta$ is global if β is a prompt without **previous** or **next**. For global temporal conditions the $sat()$ predicate is not enough. This is because the satisfiability of these temporal conditions does not depend on any specific time moment. They are either satisfied on the whole timed path or not. For instance, given a global temporal condition $cond3$ as "α **at** c $unit$ **after** β," $p, t \models cond3$ holds if for any time moment v such that β occurs at v and $v + c \leq h(p)$, there is a time moment u=$v + c$ such that α occurs at u. To check the satisfiability of $cond3$, we need to check the satisfiability of a local temporal condition $cond3'$ = "α **at** c $unit$ **after**" on all check points when β occurs. If $cond3'$ is satisfied on a check point CP, we say $cond3$ is *partially* satisfied at CP. I introduce the *l_sat* $(cond3, arguments, CP)$ predicate to represent the statement "$cond3$ is partially satisfied at the check point CP." The satisfiability of a global temporal condition is based on its *partial* satisfiability at all check points.

For the global temporal condition $cond3 = \alpha$ **at** c $unit$ **after** β, $cond3$ is satisfied on a timed path p if $cond3$ is partially satisfied at all check points on p. To implement this condition, the $\Pi(TC(\Delta), h)$ program includes the following rules:

$$\textit{l_sat}\ (cond3, \alpha, \beta, CP1)\text{:-}\ not\ \textit{-l_sat}\ (cond3, \alpha, \beta, CP1), chp(CP1). \tag{6.55}$$
$$\textit{-l_sat}\ (cond3, \alpha, \beta, CP1)\text{:-}\ prompthappen(\beta, CP1), chp(CP1),$$
$$not\ prompthappen(\alpha, CP2),$$
$$time(CP1) + c\ \$ == time(CP2),$$
$$chp(CP2), horizon\ \$ >= time(CP1) + c. \tag{6.56}$$

$$sat(cond3, \alpha, \beta, CP)\text{:- } chp(CP),$$
$$\{\textit{-l\_sat } (cond3, \alpha, \beta, CP1) : chp(CP1)\}0. \tag{6.57}$$
$$\textit{-sat } (cond3, \alpha, CP)\text{:- } chp(CP),$$
$$1\{\textit{-l\_sat } (cond3, \alpha, \beta, CP1) : chp(CP1)\}. \tag{6.58}$$

The predicate *l_sat* $(cond3, \alpha, \beta, CP1)$ states that given a check point $CP1$ such that β occurs at $time(CP1)$, there is α that occurs at $time(CP1)+c$, c time units after $time(CP1)$. I use the predicate *-l_sat* $(cond3, \alpha, \beta, CP1)$ to represent that β occurs at $time(CP1)$ but α does not occur at "$time(CP1) + c$" even if this is possible before the horizon is met. The predicate $sat(cond3, \alpha, \beta, CP)$ means that if *-l_sat* $(cond3, \alpha, \beta, CP1)$ does not hold for any check point, which means there are no check points on which $cond3$ is violated, then $cond3$ is satisfied. If there is at least one such check point, then the temporal condition is violated.

Given a temporal constraint of the form

$$\textbf{if } A_1 \textbf{ and } \ldots \textbf{ and } A_k, \textbf{then } B_1 \textbf{ or } \ldots \textbf{ or } B_m;$$

$\Pi(TC(\Delta), h)$ includes the following rule to check that for each check point, the temporal constraint is satisfied.

$$\text{:- } sat(A_1, args, CP), \ldots, sat(A_k, args, CP),$$
$$\textit{-sat } (B_1, args, CP), \textit{-sat } (B_m, args, CP),\ chp(CP). \tag{6.59}$$

This rule means that: for each sufficient check point CP, if all the temporal conditions $A_1, \ldots, A_k$ are satisfied on CP, then at least one of $B_1, \ldots, B_m$ shall be satisfied on CP as well.

Translation of Duration Specification

For each duration specification:

$$\textbf{duration } Action\ x\ unit \tag{6.60}$$

$\Pi(TC(\Delta), h)$ includes the following rules:

$$dur(Action, x). \tag{6.61}$$

$$sat(cCom, Action, S) \text{:- } dur(Action, x), horizon\$ >= time(S) + x, prompthappen(com(Action), S), chp(S). \tag{6.62}$$

$$sat(cTer, Action, S) \text{:- } next(prompthappen(ter(Action), S1), S), time(S1)\$ == time(S) + x, chp(S), dur(Action, x). \tag{6.63}$$

$$\text{:- } sat(cCom, Action, S), check(S), not\ sat(cTer, Action, S). \tag{6.64}$$

$$\text{:- } dur(Action, x), prompthappen(com(Action), S), next(prompthappen(ter(Action), S1), S), time(S1)\$! == time(S) + x. \tag{6.65}$$

where $cCom$ and $cTer$ are IDs $TeAL2ASP$ assigns for temporal conditions. The rules above represent a temporal constraint such that: for each $\mathbf{commence}\ Act$, there must be the next occurrence of $\mathbf{terminate}\ Act$ at x units later. These rules ensures that on a valid timed path, every action that can end must end.

Based on **Theorem 3**, given a $TeAL$ theory Δ, answer sets of the program $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$ represent valid paths of Δ. The translation of $\Pi(TC(\Delta), h)$ ensures that the time assignments of check points ($time(C_i) = x$) in an answer set of $\Pi(\Delta^N)_{(h,n)}$ represent the temporal information in a valid timed path.

Theorem 4 *Let Δ denote a $TeAL$ theory, Δ^N its normalized theory, h and n integers such that $0 < n \leq h$. Let A be an answer set of the program $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$. Let $p = \langle 0; s_0, t_0, pr_0; \ldots, s_{n-1}, t_{n-1}, pr_{n-1}; s_n, t_n \rangle$ be a timed path such that*

– *for every fluent f, $f \in s_i$ if and only if $holds(f, i) \in A$*

– *for every prompt* pr, $pr \in pr_i$ *if and only if* $happen(pr, i) \in A$

If p *is a valid timed path of* Δ^N*, then the program* $A \cup \Pi(TC(\Delta), h)$ *has an answer set* B *such that for every* i, $0 \leq i \leq n$, *there is* j, $1 \leq j \leq n$, $statechp(j, i) \in B$, *and* $time(j) = t_i$.

If the program $A \cup \Pi(TC(\Delta), h)$ *has an answer set* B *such that for every* $statechp(j, i) \in B$ *there is* $t_i = time(j)$, *then* $p = \langle 0; s_0, t_0, pr_0; \ldots, s_{n-1}, t_{n-1}, pr_{n-1}; s_n, t_n \rangle$ *is a valid timed path of* Δ^N.

Given a $TeAL$ theory Δ, $TeAL2ASP$ reads it and generates a program that is ready to be processed by $clingcon$. The translation introduced in this chapter ensures that the answer sets generated by $clingcon$ represent valid timed paths of Δ. If there is at least one answer set, then we can say that there is a valid timed path, and Δ is consistent within the given horizon. If $clingcon$ cannot find any answer set, then Δ is inconsistent.

Proof. This proof exploits the properties of constraint answer set [35] and splitting set [63]. The concepts of constraint answer set and splitting set are given below.

The definitions of constraint logic program and constraint answer set [35] are critical tools for the proof because a grounded $clingcon$ program $\Pi(\Delta^N)_{(h,n)}$ is a constraint logic program. Let $Atom_R$ denote a set of propositional atoms. Let Var_C denote a set of constraint variables with respective domain D such that each constraint variable v_c in Var_C has an associated domain $dom(v_c) \subseteq D$. Let $Atom_C$ denote a set of constraint atoms, where each constraint atom is an expression that includes constraint variables. Each constraint atom specifies which evaluation of the constraint variables makes this constraint atom true or false. A grounded logic program P over $Atom_R \cup Atom_C$ is a constraint logic program if for each $r \in P$ there is $head(r) \in Atom_R$.

For a constraint logic program P over $Atom_R \cup Atom_C$ and an assignment $Assign$: $Var_C \rightarrow D$, the constraint reduct P^{Assign} is the set of original rules simplified by taking into account the values of the constraint atoms, where the values of the constraint atoms

are determined by $Assign$. The logic program P^{Assign} is a regular logic program because it does not contains any constraint variables. A set $X \subseteq Atom_R$ is a constraint answer set of P with respect to $Assign$ if and only if X is an answer set of P^{Assign} [35].

According to the definition of constraint logic program, a grounded $clingcon$ program $\Pi(\Delta^N)_{(h,n)}$ is a constraint logic program because its variables $time(CP_i)$ are constraint variables with domain $[0, h]$. Let E denote the assignments $time(CP_i)\$ == int_i$, where $0 \leq int_i \leq h$. We have: a set Ans_E is a constraint answer set of $\Pi(\Delta^N)_{(h,n)}$ with respect to E if and only if Ans_E is an answer set of $\Pi(\Delta^N)^E_{(h,n)}$, the constraint reduct of $\Pi(\Delta^N)_{(h,n)}$ with respect to E. Because only $\Pi(TC(\Delta), h)$ contains constraint variables, $\Pi(\Delta^N)^E_{(h,n)}$ is equal to $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n) \cup \Pi(TC(\Delta), h)^E$, where $\Pi(TC(\Delta), h)^E$ is the constraint reduct of $\Pi(TC(\Delta), h)$.

A grounded program $\Pi(\Delta^N)^E_{(h,n)}$ contains two parts, $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$ and $\Pi(TC(\Delta), h)^E$. The timed paths defined by $\Pi(\Delta^N)^E_{(h,n)}$ are based on the paths defined by the answer set of $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$. The concept of splitting set can be used to describe the relationship between $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$ and $\Pi(TC(\Delta), h)^E$.

A splitting set of a grounded logic program P is a set of atoms U such that, for every rule $r \in P$, if $head(r) \cap U \neq \emptyset$ then all the atoms in r are included in U. The set of rules whose all atoms are included in the splitting set U is the *bottom* of P, written as $b_U(P)$. The set of other rules in P, $P \backslash b_U(P)$, is the *top* of P with respect to U, written as $t_U(P)$. Let X denote an answer set of $b_U(P)$. For the set of rules r in $t_U(P)$ such that their negative atoms $neg(r)$ are not elements of X, we define

$$Rule(t_U(P), X) = \{r' \mid head(r') = head(r), pos(r') = pos(r) \backslash X, neg(r') = neg(r)\}$$

Let Y denote an answer set of $Rule(t_U(P), X)$. The pair $\langle X, Y \rangle$ is a solution to P with respect to U. A set Ans is an answer set of P if and only if $Ans = X \cup Y$ for some solution $\langle X, Y \rangle$ to P with respect to U [63].

Let L denote the set of ground atoms in $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$. Based on the definition of a splitting set, L splits $\Pi(\Delta^N)^E_{(h,n)}$, and $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$ is the *bottom* of $\Pi(\Delta^N)^E_{(h,n)}$ with respect to the splitting set L, written as

$$b_L(\Pi(\Delta^N)^E_{(h,n)}) \ = \ \Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$$

The *top* of $\Pi(\Delta^N)^E_{(h,n)}$ relative to L is $\Pi(TC(\Delta), h)^E$.

An answer set of $\Pi(\Delta^N)^E_{(h,n)}$, Ans_E, is $A \cup B$ such that A is an answer set of the program $\Pi(\Sigma(\Delta^N)) \cup \Pi(AD(\Delta^N), n)$, B is the answer set of $e_L(E, A)$, where

$$\begin{aligned} e_L(E, A) \ &= Rule(\Pi(\Delta^N)^E_{(h,n)} \backslash b_U(\Pi(\Delta^N)^E_{(h,n)}), A) \\ &= Rule(\Pi(TC(\Delta), h)^E, A) \end{aligned}$$

and A and B are consistent ($A \cup B$ does not contain atoms a and -a at the same time). Based on the definition of $e_L(E, A)$, $A \cup B$ is an answer set of $A \cup \Pi(TC(\Delta), h)^E$.

With the definitions of constraint answer set and splitting set, we have: $\Pi(\Delta^N)_{(h,n)}$ has a constraint answer set $A \cup B$ with respect to E if and only if B is an answer set of $e_L(E, A)$.

Lemma 5 implies that given a timed path p with time assignment E, if p is valid, then $e_L(E, A)$ has an answer set which satisfies the condition in Theorem 4. Similarly, Lemma 6 implies that given a time assignment E, if $e_L(E, A)$ has an answer set, then the timed path that satisfies the condition in Theorem 4 is valid. In conclusion, $e_L(E, A)$ has an answer set if and only if there is a valid timed path p.

We have: $\Pi(\Delta^N)_{(h,n)}$ has a constraint answer set if and only if there is a valid timed path p. □

It follows that to complete the proof, it suffices to prove Lemmas 5 and 6.

Lemma 5 *If p is a valid timed path, then $e_L(E, A)$ has an answer set which satisfies the condition in Theorem 4.*

Proof. Let p denote a valid timed path. Based on the **Theorem 3**, a timed path

$$p = \langle 0; s_0, pr_0, t_0; s_1, pr_1, t_1; \ldots; s_{n-1}, pr_{n-1}, t_{n-1}; s_n, t_n \rangle$$

is a valid path if and only if $\Pi(AD(\Delta^N), n)$ has an answer set A such that: for any $i \in [0, n]$, fluent $f \in s_i$ if and only if $holds(f, i) \in A$, and prompt $pr \in pr_i$ if and only if $happen(pr, i) \in A$. Let $M(e_L(E, A))$ denote a model of $e_L(E, A)$.

Given a temporal condition C, if $p, t \models C$, then based on the rules (6.38) and (6.39), for each i such that $0 \leq i < n$, the assignment of constraint variables, E, ensures that there is one and only one atom $statechp(j, i)$ in $M(e_L(E, A))$.

Based on the rules (6.40) and (6.41), for each $holds(f, i)$ and $happen(pr, i)$

$$\begin{aligned} \mathit{fluentholds}(f, j) &\in M(e_L(E, A)) \\ prompthappen(pr, j) &\in M(e_L(E, A)) \end{aligned}$$

Based on the rules (6.42), (6.43) and (6.44)

$$chpAt(CP, C) \in M(e_L(E, A))$$

Let C be α $timeComp$ $timeMod$ β. If $p \models C$, then for any $happen(\beta, i) \in A$, $statechp(i', i) \in M(e_L(E, A))$, and $prompthappen(\beta, i') \in M(e_L(E, A))$, we have:

$$\begin{aligned} happen(\beta, j) &\in A \\ statechp(j', j) &\in M(e_L(E, A)) \\ prompthappen(\alpha, j') &\in M(e_L(E, A)) \end{aligned}$$

where the relation between i', j' is specified by $timeComp$ $timeMod$.

If C is a temporal condition in which β involves **previous** and **next**, then based on the rules (6.50) - (6.52), if $p, t \models C$, then

$$sat(C, \alpha, \beta, CP) \in M(e_L(E, A))$$

If $p, t \not\models C$, then

$$\textit{-sat}\ (C, \alpha, \beta, CP) \in M(e_L(E, A))$$

If C is a local temporal condition with only one prompt, then based on the rules (6.53) and (6.54), if $p, t \models C$, then

$$sat(C, \alpha, CP) \in M(e_L(E, A))$$

If $p, t \not\models C$, then

$$\textit{-sat}\ (C, \alpha, CP) \in M(e_L(E, A))$$

If C is a global temporal constraint, then based on the rules (6.55) and (6.56), for every check point CP such that $(\beta, CP) \in M(e_L(E, A))$, we have:

$$\textit{l\_sat}\ (C, \alpha, \beta, CP) \in M(e_L(E, A))$$

For every check point CP such that $(\beta, CP) \notin M(e_L(E, A))$, we have:

$$\textit{-l\_sat}\ (C, \alpha, \beta, CP) \notin M(e_L(E, A))$$

Based on the rules (6.57) and (6.58), if $p, t \models C$, then there does not exist any check point CP such that *-l_sat* $(C, \alpha, \beta, CP) \in M(e_L(E, A))$, thus we have

$$sat(C, \alpha, \beta, CP) \in M(e_L(E, A))$$

If $p, t \not\models C$, then there exists at least one check point CP such that *-l_sat* $(C, \alpha, \beta, CP) \in M(e_L(E, A))$, thus we have

$$\textit{-sat}\ (C, \alpha, \beta, CP) \notin M(e_L(E, A))$$

In conclusion, for any temporal condition C, if C is satisfied at a check point CP, then we have

$$sat(C, arguments, CP) \in M(e_L(E, A))$$

If C is violated at a check point CP, then we have

$$\textit{-sat}\ (C, \textit{arguments}, CP) \in M(e_L(E, A))$$

If a temporal constraint is satisfied on p, the rule (6.59) will not be violated.

Let $e_L(E, A)^R$ denote the reduct of $e_L(E, A)$ relative to $M(e_L(E, A))$. It means that $e_L(E, A)^R$ is the set of rules obtained from $e_L(E, A)$ by first dropping every rule r such that at least one of the negative atoms $neg(r)$ is an element of $M(e_L(E, A))$, then dropping the negative atoms from the bodies of all remaining rules.

Let C be a temporal condition $\alpha\ timeComp\ timeMod\ \beta$ in which β involves **previous** and **next**. If $p, t \models C$, then the rules (6.51) and (6.52) are dropped. The atom *-sat* $(C, \alpha, \beta, CP1)$ is dropped from the body of the rule (6.50). If $p, t \not\models C$ and the time moment indicated by β exists, then the rules (6.50) and (6.52) are dropped. The atom $prompthappen(\alpha, CP3)$ is dropped from the body of the rule (6.51). If $p, t \not\models C$ and the time moment indicated by β does not exist, then the rules (6.50) and (6.51) are dropped. The atom $next(prompthappen(\beta, CP2), CP1)$ is dropped from the body of the rule (6.52).

Let C be a temporal condition with only one prompt. If $p, t \models C$, then the rule (6.54) is dropped. The atom *-sat* $(C, \alpha, CP1)$ is dropped from the body of the rule (6.53). If $p, t \not\models C$, then the rule (6.53) is dropped. The atom $prompthappen(\alpha, CP2)$ is dropped from the body of the rule (6.54).

Let C be is a global temporal constraint. If $p, t \models C$, then the rule (6.56) is dropped. The atom *-l_sat* $(C, \alpha, \beta, CP1)$ is dropped from the body of the rule (6.55). If $p, t \not\models C$, then the rule (6.55) is dropped. The atom $prompthappen(\alpha, CP2)$ is dropped from the body of the rule (6.56).

Because $M(e_L(E, A))$ satisfies the above rules in $e_L(E, A)^R$, and the constraints in $e_L(E, A)^R$ are not violated, $M(e_L(E, A))$ is a model of $e_L(E, A)^R$.

Let us assume that for some M', a model of $e_L(E, A)^R$, there is

$$M'(e_L(E, A)) \subset M(e_L(E, A))$$

Let a denote an atom such that $a \in M(e_L(E,A))$ and $a \notin M'(e_L(E,A))$. If a is $statechp(c,s)$, then the rule (6.38) is violated. If a is $prompthappen(pr,s)$, then the rule (6.40) is violated. Similarly, if a is *fluentholds*(f,s), then the rule (6.41) is violated. If a is $chpAt(cp,c)$, then the rule (6.43) is violated. If a is $sat(c,\alpha,cp)$ or -*sat* (c,α,cp), then the rules $(6.50)-(6.54)$ (if C is local) or the rules $(6.57)-(6.58)$ (if C is global) are violated. Thus, we get a contradiction.

Consequently, $M(e_L(E,A))$ is the minimal among the models of $e_L(E,A)^R$. It follows that $M(e_L(E,A))$ is an answer set of $e_L(E,A)$. □

Lemma 6 *If $e_L(E,A)$ has answer set, then the timed path p which satisfies the condition in Theorem 4 is valid.*

Proof. If the reduced program $\Pi(\Delta^N)^E_{(h,n)}$ with respect to E has $A \cup M(e_L(E,A))$ as an answer set, then $e_L(E,A)$ has an answer set $M(e_L(E,A))$. Let C denote a temporal condition $\alpha\ timeComp\ timeMod\ \beta$.

– If $sat(c, arguments, cp) \in M(e_L(E,A))$, then for every check point cp_i, if

$$prompthappen(\beta, cp_i) \in M(e_L(E,A))$$

then there is cp_j such that $time(cp_i)$ is in relation $timeComp\ timeMod$ with $time(cp_j)$, and $prompthappen(\alpha, cp_j) \in M(e_L(E,A))$.
That is: if $p, time(cp) \models occur(\beta, time(cp_i))$, then we have

$$p, time(cp) \models occur(\alpha, time(cp_j))$$

Thus $p, time(cp) \models C$.

– If -*sat* $(c, arguments, cp) \in M(e_L(E,A))$, then there exists cp_i such that

$$prompthappen(\beta, cp_i) \in M(e_L(E,A))$$

but for any $time(cp_j)$ such that $time(cp_i)$ is in relation $timeComp\ timeMod$ with $time(cp_j)$, $prompthappen(\alpha, cp_j) \notin M(e_L(E, A))$.

That is: $p, time(cp) \models occur(\beta, time(cp_i))$, but there is no $time(cp_j)$ such that $p, time(cp) \models occur(\alpha, time(cp_j))$. Thus $p, time(cp) \not\models C$

In conclusion, if an atom $sat(c, argument, cp) \in M(e_L(E, A))$, then C is satisfied on p at time $time(cp)$. If an atom -*sat* $(c, argument, cp) \in M(e_L(E, A))$, then C is violated at time $time(cp)$.

Because there is no rule of the form (6.59) violated in $\Pi(\Delta^N)_{(h,n)}$, the corresponding temporal constraints must be satisfied on p at every check point. It means that all temporal constraints are satisfied on p and p is a valid timed path. □

Chapter 7 Empirical Studies

The approach introduced in this thesis is designed to reduce analysts' workload in the consistency checking task. It is necessary to prove that the efficiency of this semi-automated approach exceeds that of the manual approach. The cornerstone of this approach is the introduction of an intermediate level, $TeAL$, between the natural language requirements and the low-level logic formalism. It is necessary to prove that $TeAL$ is easier for analysts to use than the low-level logic formalism.

This approach divides the task of consistency checking into two phases: generating $TeAL$ theories from natural language requirements and processing the $TeAL$ theories to get results; the efficiency of both phases must be measured. Analysts' involvement is necessary in the first phase of the approach to ensure that a correct $TeAL$ theory is generated, so it is necessary to measure if $TeALGenerator$ facilitates this process by producing high quality $AlmostTeAL$ statements. The second phase is fully automated and hidden from analysts, so the correctness and the performance of the translation must tested.

This chapter addresses three research questions:

1. Is $TeAL$ more understandable than low-level logic formalisms such as ASP?
2. Does the $TeALGenerator$ facilitate the task of generating $TeAL$ theories?
3. Is the translation implemented by $TeAL2ASP$ efficient?

I performed three empirical studies to address the research questions above:

– Empirical Study 1 compared the understandability of $TeAL$ and ASP. This study provided evidence in support of $TeAL$ being easier for analysts to use than ASP.

– Empirical Study 2 evaluated the quality of the $AlmostTeAL$ statements generated by $TeALGenerator$. This study provided evidence that analysts generated $TeAL$ statements with better quality with the assistance of $AlmostTeAL$ statements.

– Empirical Study 3 tested $TeAL2ASP$ using six benchmarks. This study illustrated the efficiency of $TeAL2ASP$.

7.1 Empirical Study 1: Understandability of $TeAL$ and ASP

The key idea of my approach, introducing $TeAL$ as an intermediate level between natural language requirements and low-level logic formalisms, is based on the assumption that $TeAL$ is easier to understand than low-level logic formalisms. If analysts were able to efficiently comprehend low-level logic formalisms such as ASP, then there would be no need for the intermediate language approach. Hence, this experiment compares the ability of human analysts to comprehend formal requirements that are given in $TeAL$ and ASP.

Research Question

The intermediate language $TeAL$ provides an abstraction level between natural language requirements and the low-level logic formalism ASP. In my approach, the task of generating an ASP program for software requirements is reduced to generating a $TeAL$ theory. The improvement introduced by $TeAL$ is largely determined by the understandability of $TeAL$ and ASP. I designed this study to compare the understandability of these two formal languages. In this study I measured the understandability in two aspects: how quickly and how accurately analysts can understand the formal statements. I performed this study on a small scale (with fourteen participants). To study whether $TeAL$ is easier to read than ASP, I posit the following research questions:

– RQ1.1: Is it easier to understand $TeAL$ statements than to understand ASP statements?

– RQ1.2: Does it take analysts less time to understand $TeAL$ statements than to understand ASP statements?

Measures

In this study participants were given a set of natural language requirements and four candidate formal statements (in ASP/$TeAL$) for each requirement. The participants were asked to select all formal statements that represented the natural language requirement.

This study used one independent variable: Method (abbreviated as **M**)). There are two levels of the independent variable: ASP and $TeAL$.

The dependent variables that address RQ1.1 are: Precision ($\mathbf{Prec}$), Recall ($\mathbf{Rec}$), Understandability Score ($\mathbf{RS}$), and the time spent on each task ($\mathbf{T1}$). The measure $\mathbf{Prec}$ is defined as the percentage of correct answers that are selected while $\mathbf{Rec}$ is the percentage of selected answers that are correct.

$$\mathbf{Prec} = \frac{\#\ of\ correct\ answers\ selected}{\#\ of\ selected\ answers}$$

$$\mathbf{Rec} = \frac{\#\ of\ correct\ answers\ selected}{\#\ of\ correct\ answers}$$

The measure $\mathbf{RS}$ is a rating on a scale from 1 to 5 indicating the analyst's subjective judgment of the understandability of $TeAL$ or ASP statements (1 is not understandable and 5 is highly understandable). The measures $\mathbf{Prec}$ and $\mathbf{Rec}$ address the objective aspect of RQ1.1, and the measure $\mathbf{RS}$ addresses the subjective aspect of RQ1.1. I use $\mathbf{T1}$ to address RQ1.2.

Hypothesis

The null hypothesis for RQ1.1($H_{0RQ1.1}$) is that there is no difference in $\mathbf{Prec}$, $\mathbf{Rec}$, and $\mathbf{RS}$ between ASP and $TeAL$. The alternative hypothesis ($H_{1RQ1.1}$) is that there is a difference between these two methods. The null hypothesis for RQ1.2($H_{0RQ1.2}$) is that there is no

difference in **T1** between ASP and $TeAL$. The alternative hypothesis ($H_{1RQ1.2}$) is that there is a difference between these two methods.

Study Design

Each participant received a 30 minute training session that introduced $TeAL$ and ASP. During the training, the syntax and semantics of both languages were introduced along with examples. It should be noted that it is impossible to introduce the full details of ASP syntax and semantics in the 15 minutes allotted. Therefore, the training session did not cover all ASP concepts. I focused on providing examples of how actions, fluents, and temporal relationships are represented in ASP. But the semantics of ASP were largely omitted in the training session. Similarly, when I was introducing $TeAL$, I focused on its syntax rather than its semantics. It should be noted that the 15 minute sessions for each language covered the same topics at the same level of detail and thoroughness to ensure that no bias was introduced.

After the training session, the main study assignment was administered. Each participant received a set of natural language requirements and four formal statements (in ASP/$TeAL$) for each requirement. They were asked to select ALL formal statements that represent the natural language requirement. Participants were also asked to rate the understandability (**RS**) of ASP and $TeAL$.

Participants were asked to complete the tasks in the computer lab or from their home on their own time. They were also asked to keep a log of their activities during the completion of the task.

After completing the main study task, participants were asked to submit the results and complete a post-study questionnaire that asked for their reaction to requirement analysis and formal languages.

The study used multiple examples from one dataset: 511 Regional Real-Time Transit Information System Requirements (511 $phone$) [5]. This dataset presents the system re-

quirements for the Bay Area 511 Regional Real-Time Transit Information System (open source). These requirements are primarily focused on the performance of the 511 System and data transfers with the transit agencies.

Threats to Validity

This experiment was subject to a number of threats to validity, mitigated to the best of my ability. A threat to internal validity is the limited amount of time given to the participants to learn $TeAL$ and ASP. I was constrained by the amount of time available in the software engineering course class period. To address this, I separated the training session and the experiment into separate sessions (separate consecutive class periods). This allowed the participants more time to understand both formal languages. Also, I simplified the introduction of ASP by omitting its detailed semantics and organizing it into the presentation of actions, fluents, and temporal relationships. This introduces a bias in favor of ASP (I could have made ASP much harder for participants if I had spent more time showing the detailed semantics, etc.).

My work with student participants represented a threat to external validity. However, these students all had at least three years of background in computer science and they understood the concepts of software engineering and requirements engineering. Their background allowed them to perform small tasks of requirement analysis as well as professionals [83]. Another threat to validity deals with my use of one dataset. The 511phone dataset is a set of requirements used in a real project. But the results may still be not representative enough because the results may differ for a dataset from another domain. To address this threat one needs to repeat the experiment with datasets from other domains. The third threat to external validity is the motivation of the participants. Students were given extra credit to participate. This did not ensure that they answered all questions "seriously" or thoughtfully.

Dependent variable issues that threaten construct validity were reduced by the use of

both objective variables (**Prec**, **Rec**) and subjective variables (**RS**). Another threat to construct validity is that participants may have guessed the research hypothesis, that is, they may have assumed that $TeAL$ was the focus of the research before they worked on the main study assignment. I addressed this validity threat by not telling them that $TeAL$ is my research area.

A threat to conclusion validity is the low number of subjects in the study. The only way to overcome this threat is to repeat the work with more participants.

Results and Analysis

Here, I present the results of this study. Table 7.1 and Figure 7.1 present the results of the study question that addresses if $TeAL$ is more understandable and easier to work with than ASP.

Table 7.1: Results for **Prec**, **Rec**, and **T1**

Method	**Prec**	**Rec**	**T1**
ASP	0.79	0.92	83s
$TeAL$	0.84	0.95	60s

Table 7.1 shows the mean values of precision (**Prec**) and recall (**Rec**) and the time that participants spent on selecting correct formal statements (**T1**). The results of $TeAL$ are better than those of ASP in all aspects, though the results are very close in this part of the study. It appears that $TeAL$ has a significant advantage over ASP with respect to time (**T1**), that is, it took the participants much less time to work with $TeAL$ than to work with ASP. $TeAL$ also performed slightly better in terms of **Prec** and **Rec**. The **Prec** values show that given a natural language requirement, it is more likely for the participants to select an incorrect statement if the statement is represented in ASP. That the values of **Prec** are less than 85% in both $TeAL$ and ASP groups also illustrates that the chance of misunderstanding a formal statement cannot be ignored, no matter what target language is used. The **Rec** values for both groups are all close to 1, which means the participants managed

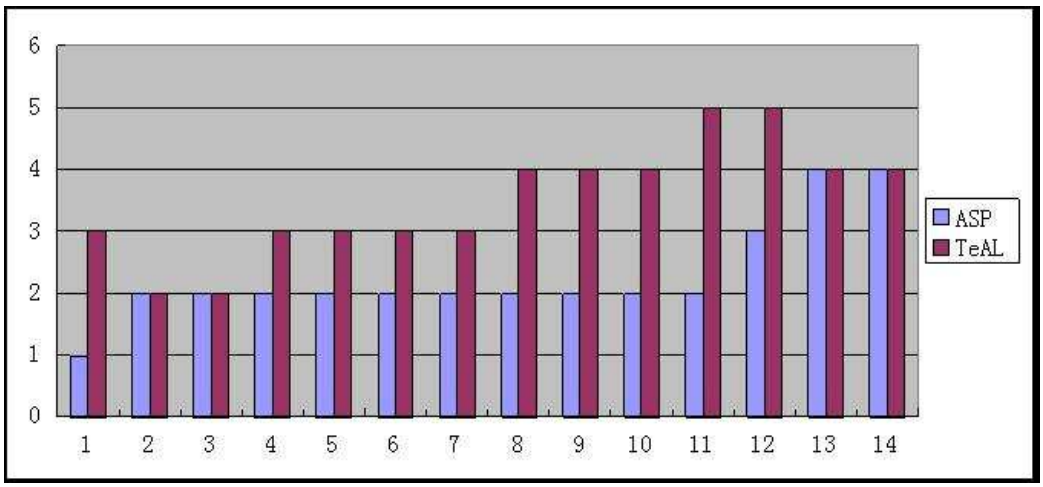


Figure 7.1: Results of Understandability Score (RS)

to find most correct formal statements for the given natural language requirements.

Figure 7.1 shows that most participants preferred to work with $TeAL$ and not ASP. As can be seen, all participants considered the understandability of $TeAL$ to be as least as good as that of ASP. Moreover, 11 out of 14 participants found ASP statements hard to understand (RS $\leq$ 2). On the contrary, 12 out of 14 participants regarded the understandability of $TeAL$ as "*neutral or better*" (RS $\geq$ 3).

To better understand what went on in my study, I conducted the statistical $t\text{-}test$ on $\mathbf{T1}$, $\mathbf{Prec}$, and $\mathbf{Rec}$ (Table 7.2). I report the standard error of difference, $t\text{-}value$, $p\text{-}value$ ($alpha$ = 0.05), and the 95% confidence interval.

Table 7.2: Results of Paired $t\text{-}test$ Analysis for $\mathbf{T1}$, $\mathbf{Prec}$, and $\mathbf{Rec}$ (ASP versus $TeAL$)

Variable	N	Diff	t	p	95% CI
T1	**14**	**4.481**	**5.1484**	**0.0002**	**(13.39, 32.75)**
Prec	14	0.090	0.6397	0.5335	(-0.25,0.14)
Rec	14	0.046	0.4977	0.6270	(-0.12,0.08)

As can be seen from Table 7.2, there is a statistically significant difference between the time participants spent on ASP and on $TeAL$. However, the improvements in precision and recall are not statistically significant (bolded items represent statistically significant dependent variables).

Based on the results above, a time savings of about 25% is realized if participants understand $TeAL$ instead of ASP. This is a major benefit of working with $TeAL$ instead of ASP. It seems intuitive that it is harder to understand an ASP statement than a $TeAL$ statement. While $TeAL$ can represent temporal relationships by using keywords close to natural language, such as "$\mathbf{laterThan}$ x $second$ $\mathbf{after}$," ASP can only represent such relationships using arithmetic formulas, such as: $t1 < t2 + 5$, where $t1$ and $t2$ are two time moments. In the post-study questionnaire, most participants also told us that they liked the keywords used in $TeAL$.

Though there were practical differences in the $\mathbf{Prec}$ and $\mathbf{Rec}$ of my study, the differences were not statistically significant. Thus I have not yet validated the effectiveness of $TeAL$ for understanding formal statements. One possible explanation is that participants still made mistakes when trying to understand $TeAL$ due to several $TeAL$ statements being able to express the same meaning, such as:

$$\mathbf{if\ terminate}\ print(server, message)$$
$$\mathbf{then\ terminate}\ send(server, ACK)\ \mathbf{noLaterThan}\ 2\ second\ \mathbf{after}$$

$$\mathbf{terminate}\ send(server, ACK)$$
$$\mathbf{noLaterThan}\ 2\ second\ \mathbf{after\ terminate}\ print(server, message)$$

Thus, participants may have forced themselves to find an additional answer after they had already found all the correct ones (in the multiple choice questions).

Returning to the research questions of interest, based on the study I found that the responses are:

– RQ1.1: It is easier to understand $TeAL$ statements than to understand ASP statements. Objectively, the chance of misunderstanding reduces slightly if the target is a $TeAL$ statement. Subjectively, participants felt more comfortable reading $TeAL$ than ASP. I can reject the null hypothesis in favor of the alternative ($H_{1RQ1.1}$).

– RQ1.2: Yes. It takes analysts less time to comprehend the meaning of a $TeAL$ statement than an ASP statement. I can reject the null hypothesis in favor of the alternative ($H_{1RQ1.2}$).

7.2 Empirical Study 2: Quality of $TeALGenerator$ Outputs

As I have discussed in Chapter 5, I developed the front end $TeALGenerator$ to facilitate the process of generating $TeAL$ theories. The $TeALGenerator$ reads texts given in natural language and outputs $AlmostTeAL$ statements. $AlmostTeAL$ statements can be used by analysts to generate $TeAL$ theories. The efficiency of generating $TeAL$ theories is based on the quality of these $AlmostTeAL$ statements.

This experiment focused on whether the outputs of $TeALGenerator$ facilitate the task of generating $TeAL$ theories from natural language requirements.

Research Questions

The quality of the outputs generated by $TeALGenerator$ is critical in this approach, because it determines how many inaccuracies an analyst needs to detect and correct in order to generate $TeAL$ statements. The $TeALGenerator$ is useful if analysts are more accurate and efficient at generating $TeAL$ with the help of $AlmostTeAL$ statements.

– RQ2.1: Does $TeALGenerator$ produce outputs that improve analyst accuracy at generating correct $TeAL$ statements?

– RQ2.2: Does $TeALGenerator$ produce outputs that improve analyst efficiency at generating correct $TeAL$ statements?

RQ2.1 and RQ2.2 are important as they directly evaluate the quality of the automated method for generating $AlmostTeAL$.

Measures

This experiment uses one independent variable: Method (abbreviated as **M**). There are two levels of this variable: $TeAL$ and $ATeAL$ ($TeAL$ with the assistance of $AlmostTeAL$).

RQ2.1 addresses the accuracy of generating $TeAL$ statements. The dependent variables that address RQ2.1 are: Precision ($\mathbf{Prec}_1$), Recall ($\mathbf{Rec}_1$), and F-measure ($\mathbf{F1}$) of predicates (for example, $send$) and constraints (for example, *if. . .then*, *within next 10 seconds*); Precision ($\mathbf{Prec}_2$), Recall ($\mathbf{Rec}_2$), F-measure ($\mathbf{F2}$) of arguments of the predicates (for example, $node$, $message$, $server$ as arguments of the action $send$ and the fluent $received$), Translation Error Rate ($\mathbf{TER}$) [82], and Translation Difficulty Score ($\mathbf{TDS}$). Table 7.3 shows the abbreviations and ranges of these variables.

Table 7.3: Dependent Variables

Variable	Abbr	Scale
Predicate Precision	$\mathbf{Prec}_1$	[0,1]
Predicate Recall	$\mathbf{Rec}_1$	[0,1]
Predicate F measure	$\mathbf{F1}$	[0,1]
Argument Precision	$\mathbf{Prec}_2$	[0,1]
Argument Recall	$\mathbf{Rec}_2$	[0,1]
Argument F measure	$\mathbf{F2}$	[0,1]
Translation Error Rate	$\mathbf{TER}$	[0,1]
Translation Difficulty Score	$\mathbf{TDS}$	{1,2,3,4,5}
Time	$\mathbf{T}$	time

A major contribution of $TeALGenerator$ is the ability to identify predicates ($Pred$) and temporal relationships ($Temp$). The basic structure of $TeAL$ statements is represented by these two types of information. The measure $\mathbf{Prec}_1$ is defined as the percentage of correct predicates and temporal relationships that are written, while the measure $\mathbf{Rec}_1$ is the percentage of written predicates and temporal relationships ($Pred/Temp$) that are correct.

$$\mathbf{Prec}_1 = \frac{\#\ of\ correct\ Pred/Temp\ written}{\#\ of\ Pred/Temp\ written}$$

$$\mathbf{Rec}_1 \;=\; \frac{\#\,of\;correct\;Pred/Temp\;written}{\#\,of\;correct\;Pred/Temp}$$

The measure $\mathbf{F1}$ is a harmonic mean of $\mathbf{Prec}_1$ and $\mathbf{Rec}_1$:

$$\mathbf{F1} \;=\; \frac{2 * \mathbf{Prec}_1 * \mathbf{Rec}_1}{\mathbf{Prec}_1 + \mathbf{Rec}_1}$$

The above formula gives equal importance to $\mathbf{Prec}_1$ and $\mathbf{Rec}_1$.

Identifying the arguments of the predicates is another important task of the front end $TeALGenerator$ because arguments are also necessary for correct $TeAL$ statements. For instance, given an action $send$, the front end needs to identify the arguments representing the sender, the object that is sent, and the destination.

Similar to the measures above, $\mathbf{Rec}_2$ defines the percentage of written arguments that are correct, $\mathbf{Prec}_2$ defines the percentage of correct arguments that are written, and $\mathbf{F2}$ is the harmonic mean of $\mathbf{Prec}_2$ and $\mathbf{Rec}_2$.

I use $\mathbf{TER}$ to measure how close a generated $TeAL$ statement is to the correct answer. The measure $\mathbf{TER}$ is an error metric for machine translation that measures the number of edits required to change a system output into a target text:

$$\mathbf{TER} \;=\; \frac{\#\,of\;edits}{average\;\#\,of\;words\;in\;target\;text}$$

where possible edits include the insertion, deletion, substitution of single words, and shifts of word sequences. I convert each $TeAL$ statement into a sequence of words to use this measure. For instance, I convert

$$received(node, msg, server)$$

$$\mathbf{within} 10\;second\;\mathbf{after\;terminate}\;send(server, msg, node)$$

into:

$$received\;node\;msg\;server\;within\;10\;second\;after\;terminate\;send\;server\;msg\;node$$

and then compare this sequence of words to the sequences given by the participants to determine how many insertions, deletions, and changes are required.

The measure **TDS** is a rating on a scale from 1 to 5 indicating the participants' subjective opinion about the difficulty of translating from natural language to $TeAL$ with/without $AlmostTeAL$.

The dependent variable that addresses RQ2.2 is the average time (**T**) spent on each question. The measure **T** evaluates the efficiency of the method.

Hypothesis

The null hypothesis for RQ2.1 ($H_{0RQ2.1}$) is that there is no difference in the $\mathbf{Prec}_1$, $\mathbf{Rec}_1$, $\mathbf{F1}$, $\mathbf{Prec}_2$, $\mathbf{Rec}_2$, $\mathbf{F2}$, **TER**, and **TDS**, between $TeAL$ and $AlmostTeAL$. The alternative hypothesis ($H_{1RQ2.1}$) is that there is a difference between the two methods. Similarly, the null hypothesis for RQ2.2 ($H_{0RQ2.2}$) is that there is no difference in the T of $TeAL$ and $AlmostTeAL$. The alternative hypothesis ($H_{1RQ2.2}$) is that there is a difference.

Study Design

The study design is very similar to the previous experiment. This study involved thirty four participants, all students in computer science courses at the University of Kentucky. A pre-study questionnaire was given to all the consenting participants to ask about their experiences in requirement engineering and formal languages. Each participant received a ten minute introduction about the background of the experiment. Participants were asked to watch a fourteen minute training video and were given a training document. The training video introduced the syntax and semantics of $TeAL$. It focused on the representation of actions, fluents, and temporal relationships in $TeAL$. The video included $AlmostTeAL$ as well. The training document covered everything in the video. The participants were required to watch the video or read the document before the main study task.

I broke the participants into two groups based on their experience in requirement engineering and formal languages. I randomly divided the participants of each experience level into two groups of the same size. One group wrote $TeAL$ statements with the help of

$AlmostTeAL$, another group wrote $TeAL$ statements but did not have $AlmostTeAL$ statements.

After the introduction, the main study assignment was administered. Each participant received a set of eight questions during the main study task. For each, they were instructed to write down the corresponding $TeAL$ statement of the given natural language requirements (with/without $AlmostTeAL$). Participants were asked to complete the tasks in the classroom. They were also asked to record the time they spent on each question.

After completing the main study task, participants were asked to submit the results and complete a post-study questionnaire that asked for their reaction to $TeAL$, $AlmostTeAL$, and the whole experiment process.

The study used examples from two datasets: 511phone system [5], which was used in the previous experiment, and CM1 [1]. The CM1 dataset is a requirement document produced by NASA for one of its science instruments. The document was released by NASA for use by the software engineering research community.

Threats to Validity

Most of the threats to validity are the same as for the previous experiment. For instance, the limited amount of time given to the participants to learn $TeAL$ is a threat to internal validity. As a mitigation, I separated the training session and the experiment into separate sessions. I also used a training video and training document, which the participants watched or read on their own time.

As in the previous experiment, dependent variables introduce a threat to construct validity. I use both objective variables ($\mathbf{Prec}_1$, $\mathbf{Rec}_1$, $\mathbf{F1}$, $\mathbf{Prec}_2$, $\mathbf{Rec}_2$, $\mathbf{F2}$) and subjective variables ($\mathbf{TDS}$) to address this threat. Another threat to construct validity is that participants may have assumed that the $ATeAL$ method is supposed to perform better than the $TeAL$ method. I addressed this threat by dividing participants into two groups; each participant used only one method.

My work with student participants introduced the same threat to external validity as in the previous experiment. However, the students also had the same background as the participants in the previous experiment. A representativeness threat to external validity was introduced because it is not possible or practical to sample requirements from every software system or even from every software domain, yet I tried to ensure that the proposed approach has been validated on representative requirements that permit generalization of results. I also mitigated this threat by using two datasets extracted from two different real projects.

Results and Analysis

Tables 7.4 - 7.7 present the results of the study question "*Does the $AlmostTeAL$ tool produce outputs that improve the accuracy of generating correct $TeAL$ statements?*" (RQ2.1). Table 7.8 addresses the study question "*Does the $AlmostTeAL$ tool produce outputs that improve the efficiency of generating correct $TeAL$ statements?*" (RQ2.2).

Table 7.4: Results for Mean Values of $\mathbf{Prec}1$, $\mathbf{Rec}1$, and $\mathbf{F1}$

M	$\mathbf{Prec}1$	$\mathbf{Rec}1$	**F1**
$TeAL$	84.13%	85.63%	84.58%
$ATeAL$	89.39%	89.28%	89.11%

Table 7.4 shows the mean values of precision ($\mathbf{Prec}_1$), recall ($\mathbf{Rec}_1$), and F-measure ($\mathbf{F1}$) associated with predicates and temporal relationships. The results of $ATeAL$ are better than those of $TeAL$ in all aspects. However, the results are very close. The values of $\mathbf{Prec}_1$, $\mathbf{Rec}_1$, and $\mathbf{F1}$ also illustrate that participants performed well in capturing the general structure of $TeAL$ statements; but the possibility of incorrect or missing predicates/constraints cannot be ignored, no matter what target language is used.

Table 7.5 shows the mean values of precision ($\mathbf{Prec}_2$), recall ($\mathbf{Rec}_1$), and F-measure ($\mathbf{F2}$). The $ATeAL$ method outperforms $TeAL$ by 30% in precision and 43% in recall. The results show that it was much more difficult for the participants who did not use

Table 7.5: Results for Mean Values of **Prec**2, **Rec**2, and **F2**

M	**Prec**2	**Rec**2	**F2**
$TeAL$	65.25%	58.31%	60.96%
$ATeAL$	**84.89%**	**83.28%**	**83.97%**

$AlmostTeAL$ to generate correct and complete sets of arguments.

Table 7.6: Results for Mean Values of **TER**

M	**TER**
$TeAL$	52.75%
$ATeAL$	**25.11%**

Table 7.6 shows the mean values of **TER**. Participants wrote better $TeAL$ with the help of $AlmostTeAL$: the number-of-edits distance from the generated $TeAL$ statements to the correct $TeAL$ was halved.

Table 7.7: Results for **TDS**

	Participants Selecting Each Score				
	1	2	3	4	5
$TeAL$	0	2	7	6	1
	0%	12%	44%	38%	6%
$ATeAL$	0	1	2	5	10
	0%	5%	11%	28%	56%

Table 7.7 shows the **TDS** results. I found that 12% of the participants in the $TeAL$ group and 5% in the $ATeAL$ group considered the process of creating $TeAL$ "*uncomfortable*" (**TDS**=2). In the $TeAL$ group, the percentage of participants who found the process "*neither comfortable nor uncomfortable*" (**TDS**=3) was 44%, while only 11% of $ATeAL$ group felt the same. I found that 38% of the $TeAL$ group considered the process "*comfortable*"(**TDS**=4), but only 6% of the $TeAL$ group felt "*very comfortable*" (**TDS**=5). In the $ATeAL$ group, the corresponding values were 28% and 56%, respectively. The median value of **TDS** was 3 ($TeAL$ group) and 5 ($AlmostTeAL$ group).

Table 7.8 shows the mean values of time spent on each question. Participants spent 40% less time with the help of $AlmostTeAL$.

Table 7.8: Results for Mean Values of **T2**

	T2
$TeAL$	282 sec
$ATeAL$	**167 sec**

Based on the results above, it is clear that $AlmostTeAL$ improves the process of generating $TeAL$ statements in terms of both accuracy and efficiency.

Table 7.9: Results of Mean Nalues for Dependent Variables ($TeAL$ versus $ATeAL$)

Variables	$TeAL$	$ATeAL$
$\mathbf{Prec_1}$	84.13%	89.39%
$\mathbf{Rec_1}$	85.63%	89.28%
F1	84.58%	89.11%
$\mathbf{Prec_2}$	65.25%	84.89%
$\mathbf{Rec_2}$	58.31%	83.28%
F2	60.96%	83.97%
TER	52.75%	25.11%
T2	282 sec	166 sec
TDS	3.38	4.33

To better understand what went on in my study, I applied the statistical t-test on all the dependent variables listed in Table 7.9. I report the standard error of difference, $t\text{-}value$, $p\text{-}value$ ($alpha$ = 0.05), and the 95% confidence interval in Table 7.10 (comparing $TeAL$ and $ATeAL$). The statistically significant results are shown in bold.

Table 7.10: Results of $t\text{-}test$ Analysis for Dependent Variables ($TeAL$ versus $ATeAL$)

Variables	Diff	t	p	95% CI
$\mathbf{Prec_1}$	0.035	1.5244	0.1372	(-0.12, 0.018)
$\mathbf{Rec_1}$	0.04	0.9206	0.3641	(-0.12, 0.04)
F1	0.034	1.3522	0.1858	(-0.12, 0.02)
$\mathbf{Prec_2}$	**0.044**	**4.4646**	**<0.0001**	**(-0.29, -0.11)**
$\mathbf{Rec_2}$	**0.037**	**6.6968**	**<0.0001**	**(-0.33, -0.17)**
F2	**0.038**	**6.1219**	**<0.0001**	**(-0.31, -0.15)**
TER	**0.068**	**4.0557**	**0.0003**	**(0.14, 0.42)**
T2	**21.279**	**5.4209**	**<0.0001**	**(72.01, 158.70)**
TDS	**0.296**	**3.2376**	**0.0028**	**(-1.56, -0.36)**

Though there were practical differences in the $\mathbf{Prec_1}$ and $\mathbf{Rec_1}$ of my study, the differ-

ences were not statistically significant. The high $\mathbf{Rec}_1$ and $\mathbf{Prec}_1$ values (84%-89%) may mean that these elements can be identified without $AlmostTeAL$. Yet the performance of $ATeAL$ was still slightly better than $TeAL$: $TeALGenerator$ correctly identified the constraints that might otherwise have been missed by the participants.

The results of $\mathbf{Prec}_2$ and $\mathbf{Rec}_2$ show that participants had a difficult time identifying arguments without $AlmostTeAL$: they missed about 40% of arguments, while 35% of the arguments they identified were incorrect. The $AlmostTeAL$ statements greatly improved both precision and recall to 83%-84%. It appears that $AlmostTeAL$ finds more correct arguments than the participants. Additionally, the missing pieces in $AlmostTeAL$, such as the blank argument in $send(node, message, \_)$, can remind participants what information to look for when they read natural language requirements. As can be seen from Table 7.10, there is a statistically significant difference between the $\mathbf{Prec}_2$, $\mathbf{Rec}_2$, and $\mathbf{F2}$ results of $TeAL$ and $ATeAL$.

The $AlmostTeAL$ statements also reduced the time spent on each question by 40% and halved the participants' error rate. The significant decrease of $\mathbf{TER}$, together with the increase of $\mathbf{Prec}_2$ and $\mathbf{Rec}_2$, lends support for the effectiveness of $ATeAL$.

Feedback from participants demonstrates that they prefer $ATeAL$ to $TeAL$. While 56% of the participants thought it was difficult to write $TeAL$ statements without any hints ($\mathbf{TDS} \leq 3$), 83% of the participants felt the availability of $AlmostTeAL$ statements provided useful information ($\mathbf{TDS} \geq 4$).

Returning to the research questions of interest, based on the study I found that the responses are:

– RQ2.1: Yes. The $AlmostTeAL$ statements generated by $TeALGenerator$ helped analysts to produce $TeAL$ with fewer errors. I can reject the null hypothesis in favor of the alternative ($H_{1RQ2.1}$).

– RQ2.2: Yes. The $AlmostTeAL$ statements generated by $TeALGenerator$ saved

time. I can reject the null hypothesis in favor of the alternative ($H_{1RQ2.2}$).

7.3 Empirical Study 3: Performance of $TeAL2ASP$

As I have discussed in Chapter 6, I designed and implemented a translator, $TeAL2ASP$ to translate $TeAL$ theories into ASP. [1] The output of the translator is passed to the ASP solver $clingcon$ for processing. The results generated by $clingcon$ illustrate if the requirements are consistent or not.

Because the process above is fully automated and analysts' involvement is not required, the efficiency of this phase depends on the time $clingcon$ takes to generate results. The time is based on the quality of the translation.

I studied the efficiency of $TeAL2ASP$ using six benchmark examples of requirement documents specifying (fragments of) real software systems. In some cases I modified these examples from their original form by varying durations of actions and including additional temporal constraints because this thesis focuses on temporal requirements. The outline of these examples and their sample requirements are given below.

CM1. This example is derived from a requirement document produced by NASA for one of its science instruments. The document was "sanitized" (hence the presence of variables rather than specific constants) and released by NASA for use by the software engineering research community [1].

– *The Control Component shall send the heart beat message to the Interface of Instrument Control Unit at an interval of E milliseconds.*

511Phone. This example is derived from a requirement document for the *Regional Real Time Transit Information System*. These requirements focus on the performance of the 511 System and the data transfers with the transit agencies. They are based on the existing procedures and features of the existing real-time system. [5].

[1] I acknowledge the assistance of David Brown with the implementation of $TeAL2ASP$.

– *If* request*, then transit agency system sends predictions and vehicle location within var1 seconds after receiving data request from the 511 System.*

MODIS. This example is derived from the open source NASA Moderate Resolution Imaging Spectroradiometer (MODIS) documents [2].

– *Each MODIS standard input data shall be produced every var1 seconds.*

UAVTCS. This example is derived from a requirement document for an Unmanned Aerial Vehicle (UAV) Tactical Control System (UAVTCS) of the US Department of Defense for the control of tactical UAVs. [4]

– *The TCS in the Normal Startup Mode shall initialize the system to the Operation State within 60 seconds from the time power is supplied.*

EasyClinic. This dataset describes a variety of artifacts from a small healthcare application. It was developed at the University of Salerno to manage a medical ambulatory [15].

– *The response time of the service shall be less than A seconds.*

iTrust. This dataset is derived from the iTrust project which involves the development of an application through which doctors can obtain and share essential patient information and can view aggregate patient data [15].

– *An HCP can reassign a previously created lab procedure to a different Lab Technician if the lab procedure is not yet in time.*

For each of these benchmarks, I created its corresponding $TeAL$ representation. I recall that the temporal constraints in my examples involve constants (parameters). Consistency of the constraints depends on specific values one chooses for these parameters. In my study, for each benchmark problem, I considered four parameter settings: (1) *underconstrained-relaxed* or UR, the temporal constraints leave much room for the system to evolve, they are

"easy" to satisfy; (2) *underconstrained-tight* or UT, the constraints are still satisfiable but they restrict significantly the ways in which the system can evolve; (3) *overconstrained-barely* or OB, the constraints are inconsistent, but a small relaxation of some of them would make them consistent; and (4) *overconstrained-much* or OM, the constraints are significantly overconstrained and no small relaxations make them consistent. Finally, I studied three values for the horizon: $h = 50, 100,$ and 200, and set the time-out limit at 7200 seconds.

Threats to Validity

This empirical study was subject to a number of threats to validity. My selection of benchmarks represented a threat to external validity. The six benchmarks used in this study were all taken from real projects, but they can not represent the software requirements in other domains. To address this threat one needs to perform this study using requirements taken from other projects. A threat to construct validity is introduced by the selection of horizon values. A horizon value must be large enough to proved evidence of consistency. But a large horizon requires more time to generate answer sets. I address this threat by estimating three horizon values: 50, 100, 200, based on the parameter settings. The way to further overcome this threat is to use more horizon values (150, 300, etc.).

Results and Analysis

Table 7.11 shows the results of my study. For each of the problems, it shows the number of constraints (**Cons**), the type of the parameter settings (UR, UT, OB and OM), and the running time. For problems that are consistent, the table also shows the number of states, n, for which the constraints were shown to be satisfiable. There were no time-outs when the theories were consistent. For overconstrained cases, the tool timed out several times (for one problem for $h = 50$, for five problems for $h = 100$, and for all problems for $h = 200$). Whenever the tool timed-out, I show in the table the last value of n for which inconsistency

Table 7.11: Results of the Study; Six Problems, Four Types of Parameter Settings

Example	**# Cons**	**Type**	**Horizon**		
			50	100	200
CM1	23	UR	395 sec, 9 states	1139 sec, 17 states	2151 sec, 34 states
		UT	429 sec, 9 states	1353 sec, 17 states	2328 sec, 34 states
		OB	5962 sec	$>$ 2 hours, 40 states	$>$ 2 hours, 37 states
		OM	5913 sec	$>$ 2 hours, 40 states	$>$ 2 hours, 37 states
511 Phone	11	UR	564 sec, 9 states	2551 sec, 18 states	3571 sec, 35 states
		UT	572 sec, 9 states	2732 sec, 18 states	3691 sec, 35 states
		OB	$>$ 2 hours, 42 states	$>$ 2 hours, 38 states	$>$ 2 hours, 36 states
		OM	$>$ 2 hours, 42 states	$>$ 2 hours, 38 states	$>$ 2 hours, 36 states
MODIS	10	UR	204 sec, 7 states	589 sec, 12 states	1787 sec, 20 states
		UT	221 sec, 7 states	594 sec, 12 states	1901 sec, 20 states
		OB	4212 sec	7009 sec	$>$ 2 hours, 47 states
		OM	4204 sec	6878 sec	$>$ 2 hours, 47 states
UAVTCS	13	UR	681 sec, 9 states	1677 sec, 17 states	4104 sec, 33 states
		UT	696 sec, 9 states	1783 sec, 17 states	4143 sec, 33 states
		OB	5813 sec	$>$ 2 hours, 42 states	$>$ 2 hours, 35 states
		OM	5771 sec	$>$ 2 hours, 42 states	$>$ 2 hours, 35 states
iTrust	12	UR	606 sec, 7 states	1574 sec, 13 states	3945 sec, 24 states
		UT	601 sec, 7 states	1591 sec, 13 states	4043 sec, 24 states
		OB	6042 sec	$>$ 2 hours, 37 states	$>$ 2 hours, 25 states
		OM	5906 sec	$>$ 2 hours, 37 states	$>$ 2 hours, 25 states
Easy Clinic	10	UR	306 sec, 8 states	1025 sec, 14 states	2775 sec, 29 states
		UT	323 sec, 8 states	1236 sec, 14 states	2834 sec, 29 states
		OB	6194 sec	$>$ 2 hours, 38 states	$>$ 2 hours, 31 states
		OM	6275 sec	$>$ 2 hours, 38 states	$>$ 2 hours, 31 states

was successfully demonstrated.

As mentioned in Chapter 4, the choice of the horizon h may affect my confidence in the determination that the requirements are consistent, and in general the larger the horizon, the stronger the evidence of consistency. However, there is a flip side to this observation. As the results show, the larger the horizon, the more computationally intensive the task of computing answer sets become. This is because the number of possible values for the number of states grows with h. Estimating a value for the number of states with which the constraints are consistent is difficult. So my tool considers all of them in turn starting with $n = 1$. If *clingcon* finds an answer set, I can say that the $TeAL$ theory does not contain inconsistency within the horizon and terminate. Otherwise, I proceed to the next value of

n or terminate (and declare inconsistency) if $n = h$.

My results also show that if a $TeAL$ theory is consistent, the consistency could be established within the time limit imposed (even for $h = 200$). This is a strong indication of the practical potential of my tool.

The situation is different when a theory is inconsistent. It takes a long time for the tool to determine inconsistency. The reason is obvious and related to the discussion above. If a $TeAL$ theory is inconsistent, then for ***each*** number of states, n, $1 \leq n \leq h$, $clingcon$ will attempt to determine consistency (that is, find an answer set) and eventually fail. However, especially when n is large, the grounding bottleneck reappears (variables representing states have to be instantiated). This makes it hard for $clingcon$ to handle large values of n.

The results show that changing the parameter combinations from UR to UT does not affect the time for computing answer sets. Similarly, there seems to be no such effect when I change from OB and OM (but here I have fewer data points to draw conclusions).

The study also shows that the number of constraints in $TeAL$ theories has much less of an effect on the effectiveness of my tool as does the value of h. The problem with the largest number of constraints, CM1, does not turn out to be more difficult than the other problems.

Chapter 8 Conclusions and Future Work

Consistency checking of natural language requirements continues to present challenges to analysts. However, when this problem is focused on temporal requirements, much can be done to semi-automate the process and minimize analysts' workload. Although analysts' involvement is necessary, my approach reduces the task of consistency checking to formalizing requirements. Automated tools and the formal language $TeAL$ have been created to reduce the time and effort required for formalizing.

The following represent the contributions of this dissertation toward the goal of minimizing analysts' workload of consistency checking:

1. A partially automated approach for consistency checking was developed. I focus on temporal requirements, and assume that requirements are given in natural language.

2. A formal language Temporal Action Language ($TeAL$) was created to represent natural language requirements. Experiments showed that people found $TeAL$ easier to understand than the ASP language.

3. A front-end tool was created to semi-automatically translate natural language requirements into $TeAL$. Experiments showed that this tool improved analysts accuracy and efficiency at generating $TeAL$ theories.

4. A fully automated translator from $TeAL$ to the ASP language was developed. Performance study showed that inconsistencies can be detected within reasonable time by analyzing the ASP program.

Future work includes improving the front-end tool. Future studies could develop modules of common (tacit) knowledge that I expect will improve the accuracy of the translation process, improve the efficiency of identifying related requirements, or look for methods to

improved the results based on the feedback from analysts. Future work also includes improving of the translator. One direction is reducing the time required to run the generated ASP program. Another direction is creating efficient translation from $TeAL$ to temporal logic. Last but not least, future work includes allowing analysts to identify the possible causes of inconsistency.

Bibliography

[1] CM-1 Dataset PROMISE Website, http://promisedata.org/promised/trunk/promisedata.org/data/cm1-maintain/cm1-maintain.txt, accessed: 2013-4-18

[2] Modis science data processing software requirements specification version 2, sdst-089, gsfc sbrs (November 1997), http://www.fas.org/irp/program/collect/uav_tcs.htm

[3] Featuring meta group: Coordinating change management and requirements for business adaptability and improved life cycle (April 2006)

[4] Uav tactical control system (May 2010), http://www.fas.org/irp/program/collect/uav_tcs.htm

[5] Regional real–time transit information system system requirements version 3.0 (2012), http://www.mtc.ca.gov/planning/tcip/Real-Time_TransitSystemRequirements_v3.0.pdf, accessed: 2013-4-18

[6] Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)

[7] Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., Rummler, A.: An exploratory study of information retrieval techniques in domain analysis. In: Software Product Line Conference, 2008. SPLC'08. 12th International. pp. 67–76. IEEE (2008)

[8] Baeza-Yates, R., Ribeiro-Neto, B., et al.: Modern Information Retrieval, vol. 463. ACM press New York (1999)

[9] Baral, C., Gelfond, M.: Reasoning agents in dynamic domains. In: Logic-based artificial intelligence, pp. 257–279. Springer (2000)

[10] Bell, T.E., Thayer, T.: Software requirements: Are they really a problem? In: Proceedings of the 2nd international conference on Software engineering. pp. 61–68. IEEE Computer Society Press (1976)

[11] Berry, D.M., Kamsties, E., Krieger, M.M.: From contract drafting to software specification: Linguistic sources of ambiguity. Online at URL http://se. uwaterloo. ca/˜ dberry/handbook/ambiguityHandbook. pdf (2003)

[12] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in computers 58, 117–148 (2003)

[13] Björkelund, A., Hafdell, L., Nugues, P.: Multilingual semantic role labeling. In: Proceedings of the Thirteenth Conference on Computational Natural Language Learning: Shared Task. pp. 43–48. Association for Computational Linguistics (2009)

[14] Cabral, G., Sampaio, A.: Formal specification generation from requirement documents. Electronic Notes in Theoretical Computer Science 195, 171–188 (2008)

[15] Capobianco, G., De Lucia, A., Oliveto, R., Panichella, A., Panichella, S.: On the role of the nouns in ir-based traceability recovery. In: Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on. pp. 148–157. IEEE (2009)

[16] Christel, M.G., Kang, K.C.: Issues in requirements elicitation. Tech. rep., DTIC Document (1992)

[17] Cimatti, A., Giunchiglia, E., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: Integrating bdd-based and sat-based symbolic model checking. In: Frontiers of Combining Systems, pp. 49–56. Springer (2002)

[18] Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)

[19] Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) 8(2), 244–263 (1986)

[20] Cleland-Huang, J., Settimi, R., Zou, X., Solc, P.: The detection and classification of non-functional requirements with application to early aspects. In: Requirements Engineering, 14th IEEE International Conference. pp. 39–48. IEEE (2006)

[21] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. The Journal of Machine Learning Research 12, 2493–2537 (2011)

[22] Craven, R., Lobo, J., Ma, J., Russo, A., Lupu, E., Bandara, A.: Expressive policy analysis with enhanced system dynamicity. In: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security. pp. 239–250. ACM (2009)

[23] Da Xu, L., Viriyasitavat, W., Ruchikachorn, P., Martin, A.: Using propositional logic for requirements verification of service workflow. Industrial Informatics, IEEE Transactions on 8(3), 639–646 (2012)

[24] De Marneffe, M.C., MacCartney, B., Manning, C.D., et al.: Generating typed dependency parses from phrase structure parses. In: Proceedings of LREC. vol. 6, pp. 449–454 (2006)

[25] De Marneffe, M.C., Manning, C.D.: Stanford typed dependencies manual. URL http://nlp. stanford. edu/software/dependencies manual. pdf (2008)

[26] Deeptimahanti, D.K., Babar, M.A.: An automated tool for generating uml models from natural language requirements. In: Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on. pp. 680–682. IEEE (2009)

[27] Doherty, P., Gustafsson, J., Karlsson, L., Kvarnström, J.: Tal: Temporal action logics language specification and tutorial. Linköping Electronic Articles in Computer and Information Science 3 (1998)

[28] Durán, A., Ruiz, A., Bernárdez, B., Toro, M.: Verifying software requirements with xslt. ACM SIGSOFT Software Engineering Notes 27(1), 39–44 (2002)

[29] East, D., Truszczyński, M.: Predicate-calculus-based logics for modeling and solving search problems. ACM Transactions on Computational Logic (TOCL) 7(1), 38–83 (2006)

[30] Emerson, E.A., Halpern, J.Y.: Decision procedures and expressiveness in the temporal logic of branching time. Journal of computer and system sciences 30(1), 1–24 (1985)

[31] Fliedl, G., Kop, C., Mayr, H.C., Winkler, C., Weber, G., Salbrechter, A.: Semantic tagging and chunk-parsing in dynamic modeling. In: Natural Language Processing and Information Systems, pp. 421–426. Springer (2004)

[32] Fogarty, W.M.: Investigation Report: Formal Investigation into the Circumstances Surrounding the Downing of Iran Air Flight 655 on 3 July 1988. Department of Defense (1988)

[33] Fuchs, N.E., Kaljurand, K., Kuhn, T.: Attempto controlled english for knowledge representation. In: Reasoning Web, pp. 104–124. Springer (2008)

[34] Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver. In: Logic Programming and Nonmonotonic Reasoning, pp. 260–265. Springer (2007)

[35] Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Logic Programming, pp. 235–249. Springer (2009)

[36] Gelfond, M., Inclezan, D.: Yet another modular action language. In: Proceedings of the Second International Workshop on Software Engineering for Answer Set Programming. pp. 64–78 (2009)

[37] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. vol. 88, pp. 1070–1080 (1988)

[38] Gelfond, M., Lobo, J.: Authorization and obligation policies in dynamic systems. In: Logic Programming, pp. 22–36. Springer (2008)

[39] Gildea, D., Jurafsky, D.: Automatic labeling of semantic roles. Computational linguistics 28(3), 245–288 (2002)

[40] Giordano, L., Martelli, A., Schwind, C.: Reasoning about actions in dynamic linear time temporal logic. Logic Journal of IGPL 9(2), 273–288 (2001)

[41] Giunchiglia, E., Lifschitz, V.: An action language based on causal explanation: Preliminary report. In: AAAI/IAAI. pp. 623–630. Citeseer (1998)

[42] Goble, L.: The blackwell guide to philosophical logic (2001)

[43] Harel, D.: Statecharts: A visual formalism for complex systems. Science of computer programming 8(3), 231–274 (1987)

[44] Hayes, J.H., Dekhtyar, A., Sundaram, S.K., Holbrook, E.A., Vadlamudi, S., April, A.: Requirements tracing on target (retro): Improving software maintenance through traceability recovery. Innovations in Systems and Software Engineering 3(3), 193–202 (2007)

[45] Hayes, J.H., Holbrook, E.A., Raphael, I., Pruett, D.M.: Fault-based analysis: How history can help improve performance and dependability requirements for high assurance systems. In: Fifth International Workshop on Requirements for High Assurance Systems (RHAS), to be presented in Chicago, IL on November. vol. 8 (2005)

[46] Heimdahl, M.P., Leveson, N.G.: Completeness and consistency analysis of state-based requirements. In: Software Engineering, 1995. ICSE 1995. 17th International Conference on. pp. 3–3. IEEE (1995)

[47] Heitmeyer, C., Labaw, B., Kiskis, D.: Consistency checking of scr-style requirements specifications. In: Requirements Engineering, 1995., Proceedings of the Second IEEE International Symposium on. pp. 56–63. IEEE (1995)

[48] Heitmeyer, C.L.: Software Cost Reduction. Wiley Online Library (2002)

[49] Holzmann, G.J.: The model checker spin. IEEE Transactions on software engineering 23(5), 279–295 (1997)

[50] Holzmann, G.: Design and validation of computer protocols.(1991)

[51] Kern, C., Greenstreet, M.R.: Formal verification in hardware design: A survey. ACM Transactions on Design Automation of Electronic Systems (TODAES) 4(2), 123–193 (1999)

[52] Klein, D., Manning, C.D.: Accurate unlexicalized parsing. In: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1. pp. 423–430. Association for Computational Linguistics (2003)

[53] Klein, M.: An exception handling approach to enhancing consistency, completeness and correctness in collaborative requirements capture. Advances in Concurrent Engineering: CE96 Proceedings p. 73 (1996)

[54] Knapp, A., Merz, S., Rauh, C.: Model checking timed uml state machines and collaborations. In: Formal Techniques in Real-Time and Fault-Tolerant Systems. pp. 395–414. Springer (2002)

[55] Kowalski, R., Sergot, M.: A logic-based calculus of events. In: Foundations of knowledge base management, pp. 23–55. Springer (1989)

[56] Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1), 134–152 (1997)

[57] Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. Formal Aspects of Computing 11(6), 637–664 (1999)

[58] Lee, J., Palla, R.: Situation calculus as answer set programming. In: AAAI (2010)

[59] Lee, J., Palla, R.: Reformulating temporal action logics in answer set programming. In: AAAI (2012)

[60] Levy, R., Andrew, G.: Tregex and tsurgeon: Tools for querying and manipulating tree data structures. In: Proceedings of the fifth international conference on Language Resources and Evaluation. pp. 2231–2234. Citeseer (2006)

[61] Li, W., Huffman Hayes, J., Truszczynski, M.: Temporal action language (tal): a controlled language for consistency checking of natural language temporal requirements. In: NASA Formal Methods, pp. 162–167. Springer (2012)

[62] Lifschitz, V.: Answer set programming and plan generation. Artificial Intelligence 138(1), 39–54 (2002)

[63] Lifschitz, V., Turner, H.: Splitting a logic program. In: ICLP. vol. 94, pp. 23–37 (1994)

[64] Lin, F., Zhao, Y.: Assat: Computing answer sets of a logic program by sat solvers. Artificial Intelligence 157(1), 115–137 (2004)

[65] Marcus, M., Kim, G., Marcinkiewicz, M.A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., Schasberger, B.: The penn treebank: Annotating predicate argument structure. In: Proceedings of the workshop on Human Language Technology. pp. 114–119. Association for Computational Linguistics (1994)

[66] Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm, pp. 375–398. Springer (1999)

[67] Mashkoor, A., Matoussi, A.: Towards validation of requirements models. In: The Second International Conference on ASM, Alloy, B and Z-ABZ 2010. vol. 5977 (2010)

[68] McCarthy, J., Hayes, P.: Some Philosophical Problems from the Standpoint of Artificial Intelligence. Stanford University USA (1968)

[69] McMillan, K.L.: Symbolic Model Checking. Springer (1993)

[70] Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence 53(1-4), 251–287 (2008)

[71] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th annual Design Automation Conference. pp. 530–535. ACM (2001)

[72] Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25(3-4), 241–273 (1999)

[73] Niemelä, I., Simons, P.: Smodels-an implementation of the stable model and well-founded semantics for normal logic programs. In: Logic Programming and Nonmonotonic Reasoning, pp. 420–429. Springer (1997)

[74] Niemelä, I., Simons, P., Soininen, T.: Stable model semantics of weight constraint rules. In: Logic Programming and Nonmonotonic Reasoning, pp. 317–331. Springer (1999)

[75] Nikora, A.P., Balcom, G.: Automated identification of ltl patterns in natural language requirements. In: Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on. pp. 185–194. IEEE (2009)

[76] Nwana, H.S.: Software agents: An overview. The knowledge engineering review 11(03), 205–244 (1996)

[77] Perrussel, L., Charrel, P.J.: Inconsistent requirements: An argumentation view. International Journal on Artificial Intelligence Tools 11(03), 303–325 (2002)

[78] Rayson, P.: Wmatrix: A web-based corpus processing environment. (2008)

[79] Schulte, C., Lagerkvist, M., Tack, G.: Gecode. Software download and online material at the website: http://www. gecode. org (2006)

[80] Schwitter, R.: English as a formal specification language. In: Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on. pp. 228–232. IEEE (2002)

[81] Schwitter, R., Ljungberg, A., Hood, D.: Ecole–a look-ahead editor for a controlled language. EAMT-CLAW03 pp. 141–150 (2003)

[82] Snover, M., Dorr, B., Schwartz, R., Micciulla, L., Makhoul, J.: A study of translation edit rate with targeted human annotation. In: Proceedings of association for machine translation in the Americas. pp. 223–231 (2006)

[83] Tichy, W.F., Padberg, F.: Empirical methods in software engineering research. In: Software Engineering-Companion, 2007. ICSE 2007 Companion. 29th International Conference on. pp. 163–164. IEEE (2007)

[84] Toutanova, K., Klein, D., Manning, C.D., Singer, Y.: Feature-rich part-of-speech tagging with a cyclic dependency network. In: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1. pp. 173–180. Association for Computational Linguistics (2003)

[85] Van Lamsweerde, A.: Requirements engineering: From system goals to uml models to software specifications (2009)

[86] Van Lamsweerde, A., Darimont, R., Letier, E.: Managing conflicts in goal-driven requirements engineering. Software Engineering, IEEE Transactions on 24(11), 908–926 (1998)

[87] Viriyasitavat, W., Da Xu, L., Martin, A.: Swspec: The requirements ppecification language in service workflow environments. Industrial Informatics, IEEE Transactions on 8(3), 631–638 (2012)

[88] Weston, N., Chitchyan, R., Rashid, A.: A framework for constructing semantically composable feature models from natural language requirements. In: Proceedings of the 13th International Software Product Line Conference. pp. 211–220. Carnegie Mellon University (2009)

[89] Yovine, S.: Model checking timed automata. In: Lectures on Embedded Systems, pp. 114–152. Springer (1998)

Vita

Personal Data:

Name: Wenbin Li

Gender: Male

Educational Background:

- Masters in Computer Science, University of Kentucky, 2010.
- Bachelor in Computer Science, Shanghai Jiaotong University, 2006.

Research Experience:

- Ph.D. Research Field: Software Engineering and Artificial Intelligence, 2006 - Present.
  Software Verification and Validation Lab,
  Department of Computer Science, University of Kentucky, Lexington, KY, USA.

Teaching Experience:

- Teaching Assistant, 08/2006 - 05/2008.
  Department of Computer Science, University of Kentucky, Lexington, KY, USA.

Work Experience:

- Research Assistant, 08/2008 - 05/2014.
  Department of Computer Science, University of Kentucky, Lexington, KY, USA.

Publications:

- Wenbin Li, Jane Huffman Hayes, and Miroslaw Truszczynski. "Towards More Efficient Requirements Formalization: A Study," In Proceedings of 21th International

Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2015)

- Wenbin Li, David Brown, Jane Huffman Hayes, and Miroslaw Truszczynski. "Answer-Set Programming in Requirements Engineering," In Proceedings of 20th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2014)

- Hayes, J.H.; Wenbin Li; Rahimi, M., "Weka meets TraceLab: Toward convenient classification: Machine learning for requirements engineering problems: A position paper," Artificial Intelligence for Requirements Engineering (AIRE), 2014 IEEE 1st International Workshop pp.9,12, 26-26 Aug. 2014

- Wenbin Li, Hayes, J.H., Fan Yang, Imai, K., Yannelli, J., Carnes, C. and Doyle, M., "Trace Matrix Analyzer (TMA)," Traceability in Emerging Forms of Software Engineering (TEFSE), 2013 International Workshop, pp.44,50, 19-19 May 2013

- Wenbin Li, Hayes, J.H., "Traceability Challenge 2013: Query+ enhancement for semantic tracing (QuEST): Software verification and validation research laboratory (SVVRL) of the University of Kentucky," Traceability in Emerging Forms of Software Engineering (TEFSE), 2013 International Workshop, pp.95,99, 19-19 May 2013

- Ashlee Holbrook, Jane Huffman Hayes, Alex Dekhtyar, Wenbin Li pA study of methods for textual satisfaction assessment,q in Journal of Empirical Software Engineering (EMSE), February 2013, Volume 18, Issue 1, pp 139-176.

- Cleland-Huang, Jane, Yonghee Shin, Ed Keenan, Adam Czauderna, Greg Leach, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jane Huffman Hayes, and Wenbin Li. "Toward actionable, broadly accessible contests in software engineering."

In Proceedings of the 2012 International Conference on Software Engineering, pp. 1329-1332. IEEE Press, 2012. Workshop

- Wenbin Li, Jane Hayes, and Miros?aw Truszczyski. "Temporal Action Language (TAL): A Controlled Language for Consistency Checking of Natural Language Temporal Requirements." NASA Formal Methods (2012): 162-167.

- Hayes, Jane Huffman, Hakim Sultanov, Wei-Keat Kong, and Wenbin Li. "Software verification and validation research laboratory (SVVRL) of the University of Kentucky: traceability challenge 2011: language translation." In Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, pp. 50-53. ACM, 2011.

- Hayes, Jane Huffman, Wei-Keat Kong, Wenbin Li, Hakim Sultanov, Steven Alexander Wilson, Sami Taha, Jody Larsen, and Senthil Sundaram. "Software Verification and Validation Research Laboratory (SVVRL) of the University of Kentucky: Traceability Challenge," in Proceedings of 2009 Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), 2009.