2010-01-01

# Test Case Automation for Specification Patterns System and Composite Propositions

Cuauhtemoc Munoz
*University of Texas at El Paso*, cuauhtemocm@miners.utep.edu

# TEST CASE AUTOMATION FOR SPECIFICATION PATTERNS SYSTEM AND COMPOSITE PROPOSITIONS

CUAUHTEMOC MUÑOZ HERRERA

Department of Computer Science

APPROVED:

Steve Roach, Ph.D., Chair

Vladik Kreinovich, Ph.D.

Scott Starks, Ph.D.

Salamah Salamah, Ph.D.

Patricia D. Witherspoon, Ph.D.
Dean of the Graduate School

# Dedication

*A mi mamá Rosa Ma. Herrera, mi papá Cuauhtémoc Muñoz (que descansa en paz), mis hermanos Cuauhtli y Aníbal, mi tía Rosina Muñoz y mi futura esposa.*

# TEST CASE AUTOMATION FOR SPECIFICATION PATTERNS SYSTEM AND COMPOSITE PROPOSITIONS

by

CUAUHTEMOC MUÑOZ HERRERA, B.S. in Computer Science

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2010

## Acknowledgements

The past years have been filled with interesting challenges and many moments which seemed to be insurmountable. However, thanks to the continued support of my professors, I was able to get through them. The professor who I believe most helped me through thick and thin and whom I want to thank first of all is Dr. Roach. I would like to thank him for seeing potential in me and providing me with the privilege of working for him. He allowed me to get my first teaching and research experiences and provided valuable professional guidance. This help came at a crucial point in my life during which I did not have many people to count on. When I decided to go for my Master's degree, I did not have a doubt who I wanted to be my mentor. I can't imagine how things had gone had he not been there for me. For this, I am very thankful and I consider him to be a part of the core group of people that I value.

I also want to thank Dr. Salamah who helped me understand the core problem of this thesis. I really appreciate his time and patient, especially when I was struggling with my work. Thank you Dr. Salamah for all the help and support you provided me.

I also want to thank Dr. Cheon for allowing me to work on my first journal paper. I wouldn't have been able to experience the excitement of presenting the paper in an international conference if he had not helped me financially. I will always remember this unforgettable experience.

I would also like to thank Dr. Kreinovich for his support and his guidance through being his T.A. I especially want to thank him for always makings classes and meetings interesting and for always ensuring that everyone learns.

Another professor who I want to thank is Dr. Novick. Thanks to him, I was allowed to teach my first professional-level class and to learn what it means to teach as a University professor.

Also, I would like to thank Dr. Starks who did not hesitate to become part of my committee for my thesis defense. I really appreciate your enthusiasm and availability even in the most difficult situations. Thank you Dr. Starks for your patience and support.

I would also like to acknowledge the following professors and staff members at the Computer Science department for the support they've provided me with: Dr, Gates, Dr. Romero, Dr. Ward, Dr. Ceberio, Dr. Magoc, Dr. Longpré, Dr. Teller, Dr. Pinheiro, Martha Losoya, Beatriz Tarango, Professor Roy, and all the professors and staff of the CS Department. They have provided me with some interesting and also funny ways of seeing problems and situations differently.

I want to thank my family, for whom I am motivated to constantly strive towards excellence. I want to thank them all for all their effort.

Lastly, I want to thank my friends and peers for providing me with stress-free moments during all the countless hours I've spent breaking my head studying and working. Especially, I would like to thank Begoña Beorlegui, Ernesto Medina, Veronica Medina, Ivan Carrazco, Lupe Espino, Omar Ochoa, Corina Vela, Melisa Vela, Patricia Esparza, Cármen Avila, César Yeep, Lillian Torres, Neith Estrada, Walter Copenhaver, Amritam, Bhanukiran, Somdev, Shreyas, Julio Olaya, Francisco Zapata, Geoffrey Owens, and all the people I have been working with.

## Abstract

Specifying software properties is a common activity in the software development process. Software properties are often written in a natural language such as English. However, the ambiguity in natural languages makes validation and verification time-consuming and error-prone. Specifying software properties in formal languages such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) enables the use of formal verification tools such as model checkers. Nevertheless, formal languages require software developers to have a strong background in mathematics and logic.

The specification Patterns System (SPS) and Composite Propositions (CPs) use a higher-level abstraction to specify software properties formally. The SPS and CPs abstractions map to well-defined LTL formulas. These templates allow software developers to focus on the specification of software properties and not on the LTL formulas. Prospec is a software tool that uses SPS and CPs to specify software properties via a graphical user interface (GUI). The output of Prospec is an LTL formula that is derived from the corresponding specification. The algorithm for creating LTL formulas from SPS and CPs is complex, and this LTL generation must be verified. In this thesis work, a new algorithm to verify LTL formulas generated by Prospec is described. The algorithm was implemented and used to test Prospec's LTL generation by covering each of the combinations of patterns, scopes, and CPs. Results of this testing effort are discussed.

# Table of Contents

## List of Tables

# List of Figures

# Chapter 1: Introduction

Formal specification languages can help software practitioners avoid ambiguous specifications and can aid the software development process. Often, these formal languages make use of automated tools to help software developers verify software systems, decreasing the software testing time and increasing the confidence in the verification process. For example, formal languages such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) are used by model checkers to verify temporal properties of software systems [Fraser and Motawa 2006]. However, to create formal specifications in a language such as LTL requires a strong mathematical background. Furthermore, the validation of LTL formulas is difficult, even for developers experienced with formal specifications [Mondragon 2004].

Fortunately, there are available tools that enable software developers to create software specifications without being immersed in formal languages.. Prospec is a tool with a graphical user interface that was developed at the University of Texas at El Paso to help software developers create software specifications [Mondragon 2004; Salamah 2007; Vela 2009]. Software developers create the software specifications by specifying patterns, scopes, and composite propositions (CPs). Prospec generates LTL formulas automatically which allows software developers to focus on the specification process rather than the definition of LTL formulas.

Although LTL formulas are generated automatically, the LTL formulas generated by Prospec are often complex and unreadable [Mondragon 2004; Salamah 2007; Vela 2009]. Validating these LTL formulas by inspection is not feasible [Mondragon 2004].

The algorithm used to generate the LTL specifications for a subset of patterns, scopes and CPs was proven correct by Salamah [2007]. This complete algorithm was implemented by Vela [2009]. Given the difficulty in validating specifications written in LTL, we would like to ensure that

the LTL formulas generated by Prospec correctly represent the semantics of the patterns, scopes, and CPs from which they are derived. Garcia [2007] implemented the PROTEF framework for testing LTL specifications using Model Checker Based Testing [Salamah et al 2005]. This approach was used to test a tiny subset of the patterns, scopes and CPs. In Garcia's testing, test cases (both inputs and expected results) were derived by hand. The number of combinations of patterns, scopes and CPs is so large that creating test cases by hand is infeasible.

***Contributions: This thesis describes an automated testing approach that achieves a significant and rigorous level of testing of LTL generation by Prospec. The algorithm used to generate test cases and solve the testing oracle problem is novel.***

In this thesis work the PROTEF framework is extended to automatically generate test cases for Prospec's LTL generator. The *LTL Verifier* is the bridge between the LTL Generator and PROTEF. The LTL Verifier uses the LTL Generator to create LTL formulas from patterns, scopes, and CPs. The verifier creates a set of test cases for each formula. Test inputs and expected outputs are passed to PROTEF for test execution. The generation of test cases automates traditional software testing techniques, including equivalence classes and boundary values analysis.

This thesis is organized as follows: Chapter Two describes the background necessary for this work, including LTL, Prospec (with SPS and CPs), software testing, and PROTEF. Chapter Three describes the test case generation process. Chapter Four shows the implementation phase of the test case generator. Finally, Chapter Five shows the results obtained from the verification process, and describes future work.

# Chapter 2: Background

This chapter first describes LTL and provides examples of LTL specification. Next, Prospec is introduced by explaining Software Property Specifications (SPS) and Composite Propositions (CPs), the foundations on which Prospec is based. Then, software testing is explained with the focus on equivalence classes and boundary value analyses. Finally, PROTEF, a model-checker-based testing tool, is introduced as a framework for executing test cases based on execution traces.

## 2.1 Linear Temporal Logic

*Linear Temporal Logic* (LTL) is a branch of logic whose formulas express properties of infinite traces over a linear frame [Leucker and Sánchez 2008]. LTL formulas are unambiguous, and they are composed of propositions (e.g. precise statements that are either true or false), and logical operators. Similar to predicate logic [Friedman et al. 2000], LTL uses the logical operators and (&), or (|), not (!), implication (→). In addition it uses the temporal operators next (X), always (G), eventually (F), until (U), weak until (W), and release (R).

LTL formulas may be evaluated with respect to a sequence of states. Each state represents a transition in time. A sequence of states is called an *execution trace* [Lo and Chao 2008]. An execution trace is visually represented by showing the set of propositions that are true in each state. Sets of propositions are given between parentheses. If no proposition is true in a state, the state is represented by a dash. Parentheses are omitted when possible. Figure 1 shows an execution trace with proposition *P* true in states 1 and 5. State numbers are typically omitted.

```
Trace      - P - - - P - -
State      0 1 2 3 4 5 6 7
```

Figure 1: Execution trace example.

Table 1 shows trace examples of the LTL logical and temporal operators. An execution trace *satisfies* an LTL formula if the formula evaluates to true for that trace.

Table 1: LTL semantics.

| **Operator** | **Name** | **Formula** | **Example Trace** | **Counter Example** |
|---|---|---|---|---|
| ! | Not | !A | -A------- | A-------- |
| & | And | A&B | (AB)----- | ----AB--- |
| \| | Or | A\|B | A--------<br>B-------- | ----A----<br>----B---- |
| U | Until | AUB | AAAAB---- | AAA----B- |
| X | Next | XA | -A------- | --A------ |
| F | Eventually | FA | ---A----- | -------- |
| G | Always | GA | AAAAAAA | AAAA-AAA |

In LTL the future includes the present. Formulas are evaluated in the current state. A formula that does not have temporal operators must be true in the current state. For the *Next* operator $X\varphi$, the formula φ must be true in the next state. For the *Eventually* operator, $F\varphi$, formula φ must either be true in the current state, or $F\varphi$ must be true in the next state (i.e., φ must be true somewhere in the trace). For the Always operator $G\varphi$, φ must be true in every state. For the *Until* operator $\varphi U\beta$, either β is true in the current state or φ must be true and $\varphi U\beta$ is true in the next state.

LTL can help formalize software properties. For example, the typical problem of mutual exclusion where two processes *P1* and *P2* cannot be in the critical section at the same time can be represented in LTL as follows: **G!(P1 & P2),** which specifies that whenever *P1* is true, *P2* must not be true or vice versa.

Model checking is a formal verification technique that uses Temporal Logic formulas [Miller et al. 2006]. There is a variety of model checking software available such as SPIN [Holzmann 2003] and NuSMV [Cimmati et al. 2002]. The input parameters to a model checker are a Temporal Logic (e.g, LTL, or CTL) formula and a model. The model is a set of Finite State Machines (FSM) [Miller

et al. 2006]. The model checker traverses the states of the model and checks the validity of the formula in each state. When more than one FSM is present in the model, the model checker checks all possible interleavings of states. The model checker provides a mechanism determining whether a formula representing a specification is valid for a model or not.

The specification of some software properties in LTL can be a difficult task for software developers. For example, the LTL formula in Figure 2 mean that "between the start and end of a computer application, the CPU must be idle, there must be enough memory space available, and the hard disk must have enough space to allocate the application's data." However, it is not clear whether or not the LTL formula describes the specification correctly. Obtaining the result for simple LTL formulas is straightforward, but when the LTL formulas become complex, determining the truth value of the formula for a trace can be difficult.

(G ((start & ! end & F end)→ (!end U (CPU_Idle & Enough_Memory & Enough_Space))))

Figure 2: LTL formula example.

Another example is shown in Figure 3 where the LTL formula is represented in a pseudo code style [Gunter 2003].

**Specification:**
*Once the pump has been turned on, an initial pressure reading will be taken and, if the value read is not an error, it is displayed. Thereafter, provided there are no errors in pressure readings, the display will never be more than 2 seconds old.*
**LTL Formula:**
*Pump_on & !error(pressure_reading) --> Eventually(display_value = previous_pressure_reading & last_display_update_time = previous_current_time & (!error(pressure_reading) Releases(seconds(current_time – last_display_update_time) <= 2 Until(display_value = previous_pressure_reading & last_display_update_time = previous_current_time))))*

Figure 3: Specification in natural language and its LTL formula equivalence.

## 2.2 Prospec

*Prospec* is a framework capable of automatically generating LTL formulas from software property specifications [Mondragon 2004; Salamah 2007; Vela 2009]. The software property specifications that Prospec supports are based on the *Specification Patterns System* (SPS) [Dwyer et al. 1999] and *Composite Propositions* (CP) [Mondragon 2004]. The following sections provide a general description of SPS and CPs.

### 2.2.1 Specification Patterns System

SPS defines a set of templates that abstracts common software specifications from real world software systems [Dwyer et al. 1999]. The SPS defines two types of property patterns: *occurrence* and *order* patterns. Occurrence patterns describe the occurrence or existence or certain events during the execution of software systems [Dwyer et al. 1999]. Order patterns describe pairs of events during the execution of software systems [Dwyer et al. 1999]. The property patterns taxonomy is shown in Figure 4. The shaded patterns in Figure 4 are the patterns supported by Prospec.

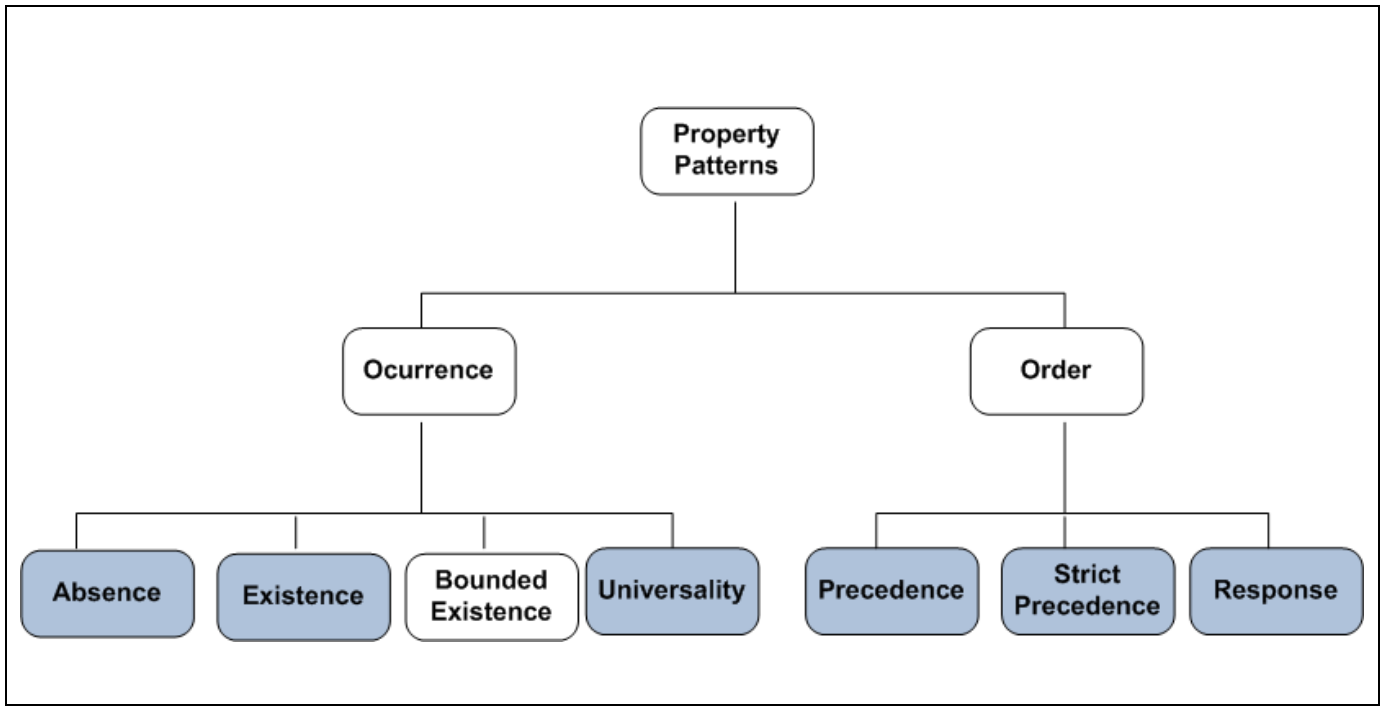

Figure 4: Software Property Specifications Taxonomy.

Scopes define the extent to which each pattern holds in an execution trace [Dwyer et al. 1999]. The scopes shown in Figure 5 are *Global*, *Before R*, *After Q*, *Between Q and R*, and *After Q until R*. Each pattern and each scope are associated with some propositions [Dwyer et al. 1999].

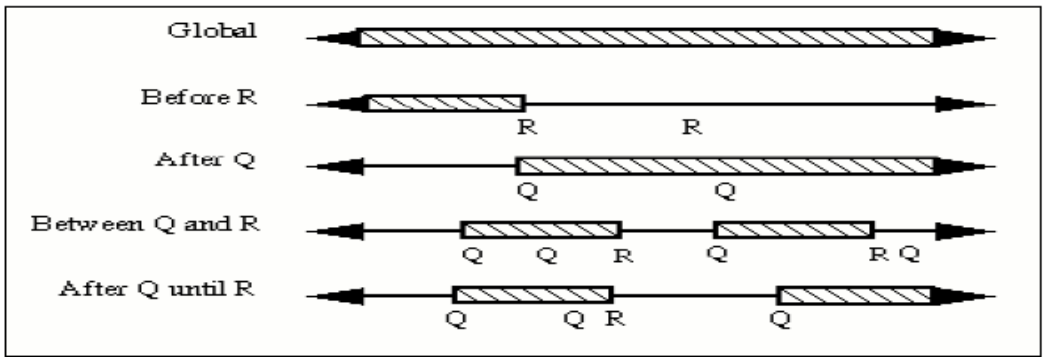


Figure 5: Software Property Specifications Scopes.

Propositions represent *events* or *conditions* that are true or false at a given state [Mondragon 2004]. For example, if a reference to the first element in a linked list is maintained by the head reference, the head reference must always exist to preserve the linked list. Thus, the existence of the head node is a necessary condition that must hold in all possible states in the program following tree initialization.

The *absence* pattern describes specific events or conditions not present in an execution trace [Dwyer et al. 1999]. The *existence* pattern describes the existence of an event or condition in an execution trace. The *response* pattern states that an event or condition must be present and followed by another condition or event. The *precedence* pattern states that some specific condition or event is necessary for another condition to occur. Finally, the *strict* precedence pattern differs from the precedence pattern in that the former does not allow the two events or conditions to hold in the same state. Table 2 shows satisfying and falsifying execution traces for the absence, existence, response, precedence, and strict precedence patterns.

Table 2: Execution traces for the absence, existence, response, precedence, and strict precedence.

| Pattern | Satisfying Execution Trace | Falsifying Execution Trace |
|---|---|---|
| Absence of P | ---------- | P---------- |
| Existence of P | --------P------- | ---------- |
| Response: (T) responds to (P) | -------P-----T----- | ------T-------P--- |
| Precedence: (T) precedes (P) | -----T---------- | -----P---------T---- |
| Strict Precedence: (T) precedes (P) | -------T---P------- | ----- (TP)---------- |

In SPS, each pattern is associated with a scope that specifies the range of program execution over which the pattern holds. The *global* scope states that a pattern must hold in the entire execution trace. The *before R* scope states that the pattern must hold up to the time the condition or event R holds. The *after Q* scope defines a pattern that holds after the given proposition Q is true. The *after Q until R* scope defines a pattern that must hold after Q and strictly before R. This scope can also continue after Q until the end of the execution trace if R is not specified. The *between Q and R* scope is similar to the scope after Q until R except that the proposition R must be present in the execution trace when Q is present. Tables 3, 4, 5, 6, and 7 shows the *absence*, *existence*, *response*, *precedence*, and *strict precedence* with the scopes *global*, *before R*, *after Q*, *after Q until R*, and *between Q and R* with sample valid and invalid execution traces. In Table 5, T responds to P; in Table 6, T precedes P, and in Table 7 T strictly precedes P.

Table 3: Absence pattern with scopes.

| Pattern | Scope | Satisfying Execution Trace | Falsifying Execution Trace |
|---|---|---|---|
| Absence | Global | ---------- | P---------- |
| Absence | Before R | ------R---------- | P-----R---------- |
| Absence | After Q | -------Q---------- | ----Q--------P------- |
| Absence | After Q until R | -------Q---------R | ---Q---P---------- |
| Absence | Between Q and R | ---Q-----R--------- | ---Q----P----R------ |

Table 4: Existence pattern with scopes.

| Pattern | Scope | Satisfying Execution Trace | Falsifying Execution Trace |
|---|---|---|---|
| Existence | Global | ----P---- | ---------- |
| Existence | Before R | P---R---------- | -------R---------- |
| Existence | After Q | -------QP---------- | ---------Q---------- |
| Existence | After Q until R | --Q-----P----R----- | -----Q------R-------- |
| Existence | Between Q and R | ------Q-----P-----R | --Q-------R---------- |

Table 5: Response pattern with scopes.

| Pattern | Scope | Satisfying Execution Trace | Falsifying Execution Trace |
|---|---|---|---|
| Response | Global | ------P------T------ | P---------- |
| Response | Before R | PT------R---------- | P------R---------- |
| Response | After Q | -----Q-----P----T-- | ----------QTP |
| Response | After Q until R | --------QPT----R | ------QTP----------R |
| Response | Between Q and R | -------QPT---- R | ------- Q---P-----R |

Table 6: Precedence pattern with scopes.

| Pattern | Scope | Satisfying Execution Trace | Falsifying Execution Trace |
|---|---|---|---|
| Precedence | Global | ---T----P--------- | PT---------- |
| Precedence | Before R | TP----R--------- | PT--------R---------- |
| Precedence | After Q | -----Q----T-----P | ----------QPT----- |
| Precedence | After Q until R | -------QTP---R | ----QPT----------R |
| Precedence | Between Q and R | ------QTP----R | -----Q----PT-----R |

Table 7: Strict Precedence pattern with scopes.

| Pattern | Scope | Satisfying Execution Trace | Falsifying Execution Trace |
|---|---|---|---|
| Strict Precedence | Global | ---T----P--------- | PT---------- |
| Strict | Before R | TP----R--------- | PT--------R---------- |

| | | | |
|---|---|---|---|
| Precedence | | | |
| Strict Precedence | After Q | -----Q----T-----P | ----------Q(PT)----- |
| Strict Precedence | After Q until R | -------QTP---R | ----Q(PT)----------R |
| Strict Precedence | Between Q and R | ------QTP----R | -----Q----PT-----R |

### 2.2.2 Composite Propositions

Composite propositions (CPs) support concurrent and sequential behavior by allowing definitions of patterns and scopes using multiple conditions and events. CPs are classified as either *events* or *conditions*. On one hand, CPs of type condition hold in one or more consecutive states, and these can represent concurrency. On the other hand a CP of type event requires a change of the values of the propositions in two consecutive states. CPs of type event can represent either synchronization or activation [Mondragon 2004]. There are 12 classes of CP, however this thesis just considers eight of these classes: *at least one (condition), at least one (event), eventually (condition), eventually (event), consecutive (condition), consecutive (event), parallel (condition), and parallel (event)*. Table 8 shows the semantics of composite propositions in LTL. Note that in Table 8 the subscripts *C* and *E* denote condition and event respectively.

Table 8: Composite Propositions in LTL.

| Syntax | Semantics in LTL |
|---|---|
| $AtLeastOne_C(\{p_1, p_2, p_3\})$ | $p_1 \mid p_2 \mid p_3$ |
| $AtLeastOne_E(\{p_1, p_2, p_3\})$ | $(!p_1 \,\&\, !p_2 \,\&\, !p_3) \,\&\, ((!p_1 \mid !p_2 \mid !p_3)\ U\ (p_1 \mid p_2 \mid p_3))$ |
| $Parallel_C(\{p_1, p_2, p_3\})$ | $p_1 \,\&\, p_2 \,\&\, p_3$ |
| $Parallel_E(\{p_1, p_2, p_3\})$ | $(!p_1 \,\&\, !p_2 \,\&\, !p_3) \,\&\, ((!p_1 \,\&\, !p_2 \,\&\, !p_3)\ U\ (p_1 \,\&\, p_2 \,\&\, p_3))$ |
| $Consecutive_C(<p_1, p_2, p_3>)$ | $(p_1 \,\&\, \mathbf{X}(\, p_2 \,\&\, \mathbf{X}(p_3)\,)$ |
| $Consecutive_E(<p_1, p_2, p_3>)$ | $(!p_1 \,\&\, !p_2 \,\&!p_3) \,\&\, ((!p_1 \,\&\, !p_2 \,\&!p_3)$ |

| | |
|---|---|
| | U ($p_1$ & !$p_2$ &!$p_3$ & X( $p_2$ &!$p_3$))) |
| Eventually$_C$(<$p_1$, $p_2$, $p_3$>) | $p_1$ & **X**(!$p_2$ U($p_2$ &X(!$p_3$ U($p_3$)))) |
| Eventually$_E$(<$p_1$, $p_2$, $p_3$>) | (!$p_1$ & !$p_2$ & !$p_3$) & ((!$p_1$ & !$p_2$ & !$p_3$) U ($p_1$ & !$p_2$ & !$p_3$ &((!$p_2$ & !$p_3$) U ($p_2$ & !$p_3$ &(!p3 U p3)))) |

All the CPs have the notion of a *beginning state* and an *ending state* [Salamah 2007]. Thus, the first state at which the composite proposition holds is considered as the beginning state, and the last state at which the composite proposition holds is the ending state [Salamah 2007]. Figure 6 shows an example of an execution trace using CPs with their respective beginning and ending states.

**Pattern:** Absence of P
**Scope:** Before R
**Composite Propositions:**
**P**: Eventually$_C$ - P1, P2, and P3
**R**: Eventually$_C$ - R1, R2, and R3

**Execution Trace:**

| | | | | | | | | | | | | | | P1 | P2 | P3 | R1 | R2 | R3 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | FALSIFIED |

**Beginning State for P:** State 14 where P1 holds since P is of type condition.
**Ending State for P:** State 16 where P3 holds for the same reason above.

**Beginning State for R:** State 17 where R1 holds.
**Ending State for R:** State 19 where R3 holds.

Figure 6: SPS Patterns and Scopes and CP Example with Conditions

*Absence of P Before R* means that no *P* must hold before *R*. In Figure 6 *P* holds before *R*, and *P* is of type *Eventually$_C$* which means that eventually *P3* must be preceded by *P2*, and *P2* must be preceded by *P1* before *R*. Therefore, the trace is falsified as *P* holds before *R* (e.g. *P1*, *P2*, and *P3* hold before *R1*). Figure 7 shows the same pattern and scope combination from Figure 6, except that the CP for *R* is of type event. The explanation for beginning states and ending states is that both the beginning and ending states must hold within the scope [Salamah 2007]. In Figure 7, the beginning

state for *R1* is state 16, the last state where all *R* propositions were false. The last state for *P* is also State 16, the state where all of P's propositions have become true. Therefore, the end of *P* does not occur before the start of *R*, and the formula is satisfied.

**Pattern:** Absence of P
**Scope:** Before R
**Composite Propositions:**
**P**: Eventually$_C$ - P1, P2, and P3
**R**: Eventually$_E$ - R1, R2, and R3

**Execution Trace:**

| | | | | | | | | | | | | | P1 | P2 | P3 | R1 | R2 | R3 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

SATISFIED

**Beginning State for P:** State 14 where P1 holds since P is of type condition.
**Ending State for P:** State 16 where P3 holds for the same reason above.

**Beginning State for R:** State 16 as this is the last state where R is false.
**Ending State for R:** State 19 where R3 holds.

Figure 7: SPS and CP Example with Events

### 2.2.3 LTL Generator

The *LTL Generator* is based on Salamah's algorithm [2007] for generating LTL formulas for SPS patterns and CPs. Given a pattern, a scope, and composite propositions, the LTL Generator automatically generates LTL formulas by using a set of templates.

Patterns, scopes, and CPs are provided by the user and translated into their corresponding LTL formula in the LTL Generator. For example, given the pattern *Absence of P (AtLeastOne$_C$)*, and the scope *Before R (AtLeastOne$_E$)*, the LTL Generator generates the corresponding LTL formula as shown in Figure 8.

```
((F  (( !R1 & !R2 & !R3) & (( !R1 & !R2 & !R3) U (R1 | R2 | R3))))
-> !((!(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) U ((P1 | P2 | P3) &
!(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 )))))
```

Figure 8: LTL Generator Formula Example

The LTL Generator is capable of automatically generating LTL formulas for all patterns, scopes, and CPs. There are infinitely many possible execution traces (as in Figure 6) that can be generated with the application of SPS and CPs. Manually generating a reasonable set of execution traces is not feasible (we will explain this with a number of possible combinations for each SPS and CP in Chapter 3). Therefore, we want to use Prospec, especially the LTL Generator, to generate LTL formulas automatically from SPS and CPs. Then, by generating test cases automatically for these LTL formulas, we want to obtain confidence in the mapping between SPS and CPs with the LTL formulas.

## 2.3 Software Testing

*Software testing* is a part of the software engineering discipline that is dedicated to validating and verifying software systems [Hamlet and Myer 1994]. Several mechanisms in software testing can help software developers verify software systems, including unit testing and system testing [Amman and Offuit 2008].

Although software testing is one of the most common techniques used for validation and verification, it is impossible to test software systems completely [Balci et al. 2002]. This is a theoretical limitation meaning that finding all errors and failures in a software system is undecidable [Balci et al. 2002]. Therefore, it is reasonable to select test cases wisely, and that can offer software developers a good coverage for verifying software systems.

### 2.3.1 Equivalence Classes and Boundary Values Analysis

Testing every input value in a program is in most cases impossible, and software developers are forced to select a smaller subset of possible input values. The new subset of input values can be partitioned into a finite number of *equivalence classes* (valid and invalid equivalence classes) [Glenford 2004]. Equivalence classes provide a mechanism for covering all input values. The

selection of test cases is obtained by partitioning the input values in such a way that if one test case fails in the equivalence class, other test cases from the equivalence class would fail. Similarly, if a test case passes in the equivalence class, other test cases from the equivalence class would pass [Glenford 2004]. Table 9 shows an example of equivalence classes using the specification from Figure 3.

Table 9: Equivalence Classes Example.

| **Input Condition** | **Valid Equivalence Class** | **Invalid Equivalence Class** |
|---|---|---|
| Switch on and switch off mechanisms for pump. | Pump is on.<br><br>Pump is off. | Other pump mechanism (i.e. switch to rotate pump). |
| Initial pump readings. | Pressure readings. | Other pump readings (i.e. temperature readings). |
| Display's longevity data intervals after reading pressure values. | < 2 seconds | > 2 seconds<br><br>< 0 seconds |

Most failures occur at the boundaries of the partitioned input values. *Boundary values analysis* produces input values at the boundaries of the partitioned input values [Hierons 2006]. Thus, combining equivalence classes and boundary values analysis can give us more confidence for finding more failures. To illustrate the importance of boundary values analysis, let us consider the specification from Figure 3 with boundary values analyses as sown in Table 10.

Table 10: Boundary Values Analyses Example.

| **Input Condition** | **Boundary Values** |
|---|---|
| Switch on and switch off mechanisms for pump. | On<br>Off |
| Initial pump readings. | 0<br>-1 or -0.009 |
| Display's longevity data intervals after reading pressure values. | 2<br>2.1<br>0<br>-0.9 |

## 2.4 PROTEF and Model-Checker-Based Testing

*Software testing automation* is a technique used in the software development process that makes use of heuristics to generate test cases automatically [Shahamiri et al. 2009]. Software testing automation is useful when there is a myriad of possible test cases that software developers need to create [Berner et al. 2005]. Automated testing techniques include automatic generation of test cases from random values [Cheon 2007], from recorded test cases [Sztipanovits 2008], from diagrams (e.g. Unified Modeling Language [Kansomkeat and Rivepiboon 2003]), or from model specifications in model-checker-based testing [Fraser and Wotawa 2006]. For example, JET is an automated testing tool that generates random test cases using a unit testing approach [Cheon 2007]. Another example of automated software testing tools is PROTEF which is a model-based testing framework dedicated to generating models automatically for the NuSMV model checker [Cimatti et al. 2002].

For example, the SPIN model checker takes a model and an LTL formula as a correctness claim (i.e. property of interest), converts the LTL formula and the model into a Büchi automata each, and computes the intersection of both Büchi automata. If the language accepted by the intersection automaton is empty, the LTL formula is not satisfied by the given model. Otherwise, if the language is nonempty, the model satisfies the LTL formula [Holzmann 1997].

Given an LTL formula and an execution trace, PROTEF generates models suitable for use by the NuSMV model checker. The model is executed by NuSMV, which determines whether the model satisfies the LTL formula. Figure 9 shows the PROTEF framework data flow diagram.

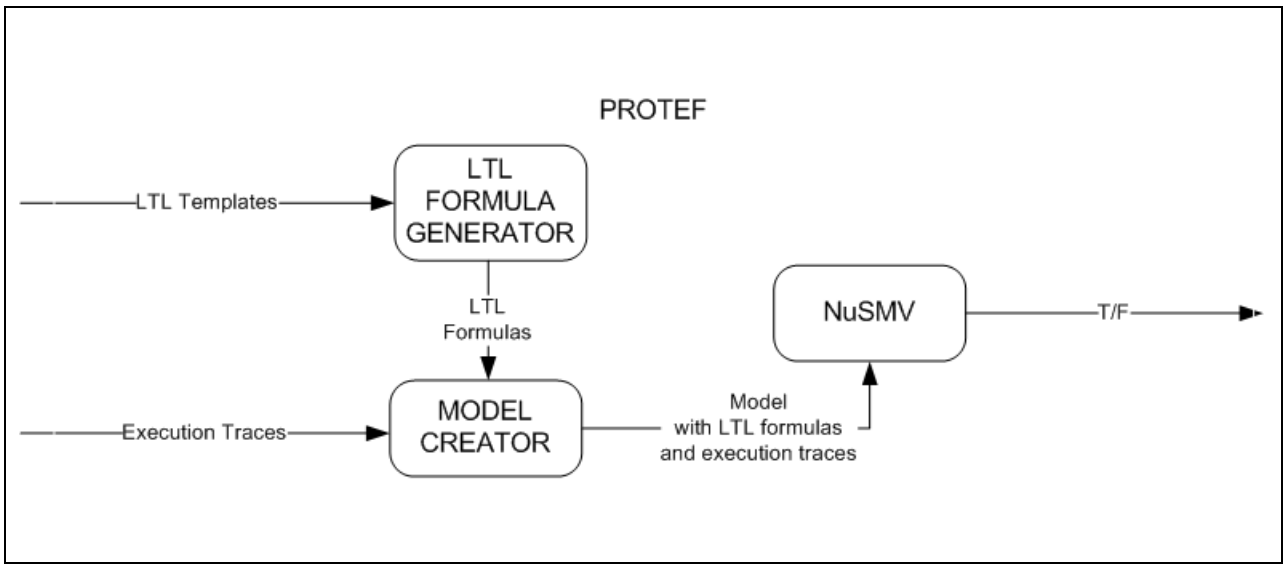


Figure 9: PROTEF Framework.

Garcia [2007] implemented a prototype LTL Formula Generator (shown in Figure 9) to show that PROTEF generates correct models from specific LTL formulas. The LTL formulas must be provided by following the LTL templates implemented by Salamah [2007]. Execution traces are manually specified by the user in plain text. The execution traces are of the form explained in previous chapters (e.g. P----P---). Also, the execution traces are created based on patterns, scopes, and CPs. Next, PROTEF creates models automatically based on the LTL formula specifications and the set of execution traces. Finally, the automatically generated models are passed to NuSMV to check if the models satisfy the LTL formula specifications.

## Chapter 3: Automated Test Case Generation for Prospec LTL Formulas

Given that formal LTL specifications may be used to verify software, it is important to ensure that the specifications are correct. The algorithm for generating LTL formulas from patterns, scopes, CPs, and propositions is complex. The complexity of the resulting LTL formulas makes it impractical to verify the formulas manually. Manual proofs of correctness for some parts of the algorithm have been constructed [Salamah 2007]. While care was taken during the implementation of the code, only a small set of manual test cases for specific patterns, scopes, and CPs has been written [Garcia 2007; Salamah 2007; Vela 2009].

One approach to achieving confidence in Prospec's LTL Generator is rigorous testing. An ambitious goal for coverage will test every combination of pattern, scope, and CP. The number of combinations is in the tens of thousands. Significant test cases for each combination can be obtained through boundary value and equivalence class analysis. To obtain even modest coverage requires hundreds of thousands of test cases. Creating these test cases manually is infeasible.

This chapter proposes an algorithm to automatically generate test cases in the form of LTL specification, execution trace, and predicted satisfiability. The LTL formulas generated from Prospec should match the description for the patterns, scopes, and CPs [Salamah 2007]. Verifying the implementation against the theoretical description with a practical implementation can provide more confidence in Prospec's LTL formula generation.

Given a pattern and a scope, the LTL Verifier generates test cases for all combinations of CPs. The output of the verification consists of a set of tests, where each test consists of (1) an informal specification containing a pattern, a scope, and a set of CPs; (2) an execution trace containing a sequence of states and values of propositional variables; and (3) the expected result: either *satisfied* or *falsified*.

Figure 10 shows the LTL Verifier system which provides the steps to generate test cases automatically.

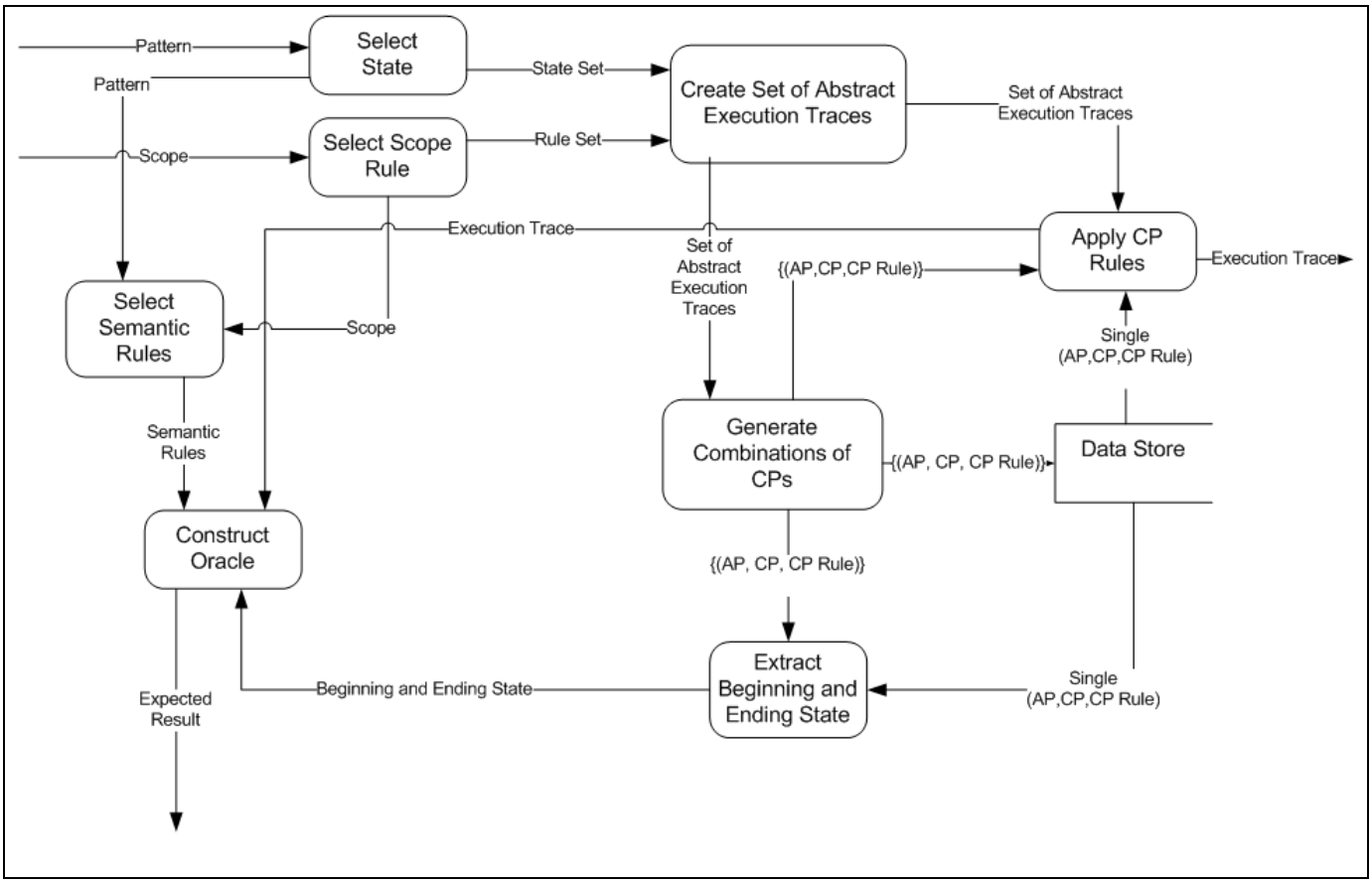


Figure 10: Data Flow Diagram of the LTL Verifier.

### 3.1 Generating Abstract Execution Traces

Abstract execution traces are generated by combining states and scope rules. These execution traces are abstract, since they do not contain information about CPs such as ending and beginning states. The set of propositions are placed in the placeholders specified by Start State, Middle State, and End State. Figure 11 shows some examples of the abstract execution traces for *Absence of P Before R*.

**Absence of P Before R**

**(P): {**P1, P2, P3, (P1P2), (P1P3), (P2P3)}

**(R): {**R1, R2, R3, (R1R2), (R1R3), (R2R3)}

**Start State:**

P1 ___ ___ ___ ___ ___ ___

P2 ___ ___ ___ ___ ___ ___

P3 ___ ___ ___ ___ ___ ___

(P1P2) ___ ___ ___ ___ ___ ___

**……..**

**(P, R): {**P1R1, P1R2, P1R3, P2R1, P2R2, P2R3, P3R1, P3R2,…., (P1R1), (P1R2), (P1R3), (P2R1), (P2R2), (P2R3), (P3R1), (P3R2),…, P1--R1, P1--R2, P1--R3, P2--R1, P2--R2, P2--R3, P3--R1, P2--R2,…, R1P1, R1P2, R1P3, R2P1, R2P2, R2P3, R3P1, R3P2,…, P1P2P3, P1P2P3R1, P1P2P3R2,…….,---}

**Start State:**

P1 R1 ___ ___ ___ ___ ___

P1 R2 ___ ___ ___ ___ ___

R3 P2 ___ ___ ___ ___ ___

**…….**

***Middle State and End State are similar to Start State.***

Figure 11: Abstract execution traces for *Absence of P Before R*.

However, generating abstract execution traces from patterns and scopes is not sufficient. The LTL Generator supports patterns, scopes, and CPs. Therefore, to fully verify the LTL Generator it is also necessary to provide combination of CPs in the execution traces.

### 3.2 Generating Combinations of Composite Propositions

Generating combination of CPs means that each proposition in an abstract execution trace can have a well-defined CP class. A class can be any of the eight CP classes supported by the LTL Generator. The process for generating combination of CPs requires abstract execution traces as input values. These abstract execution traces contain propositions (e.g. AP or abstract propositions as they do not contain information about ending and beginning states) that can be assigned a specific CP

class. The result of applying CP classes to each abstract proposition creates a new set of abstract execution traces. The new set of abstract execution traces contains CP classes for each proposition called *CP Rules*. CP Rules can be any of the eight CP classes such as $AtLeastOne_C$ or $Parallel_C$.

For example, the *Absence of P* pattern can be used with the *After L until R* scope. For each of the three propositions (P, L, and R) in the abstract execution trace there are eight possible CPs. Thus, 512 combinations are generated, as shown in Figure 12. Table 11 shows the total number of possible combinations (30,160 possible combinations) where each combination can have one or more possible abstract execution traces. For example, the total number of combinations (8) from *Absence Global* in Table 11 is obtained by assigning a CP class to each proposition present in the pattern and scope combination. If the complete definition is *Absence of P Global*, the only proposition defined is P and thus only eight possible CP classes can be assigned to this definition (e.g. $AtLeastOne_C(P)$, $AtLeastOne_E$(P), and so on). Each CP combination is stored in the *Data Store* component as shown in Figure 10.

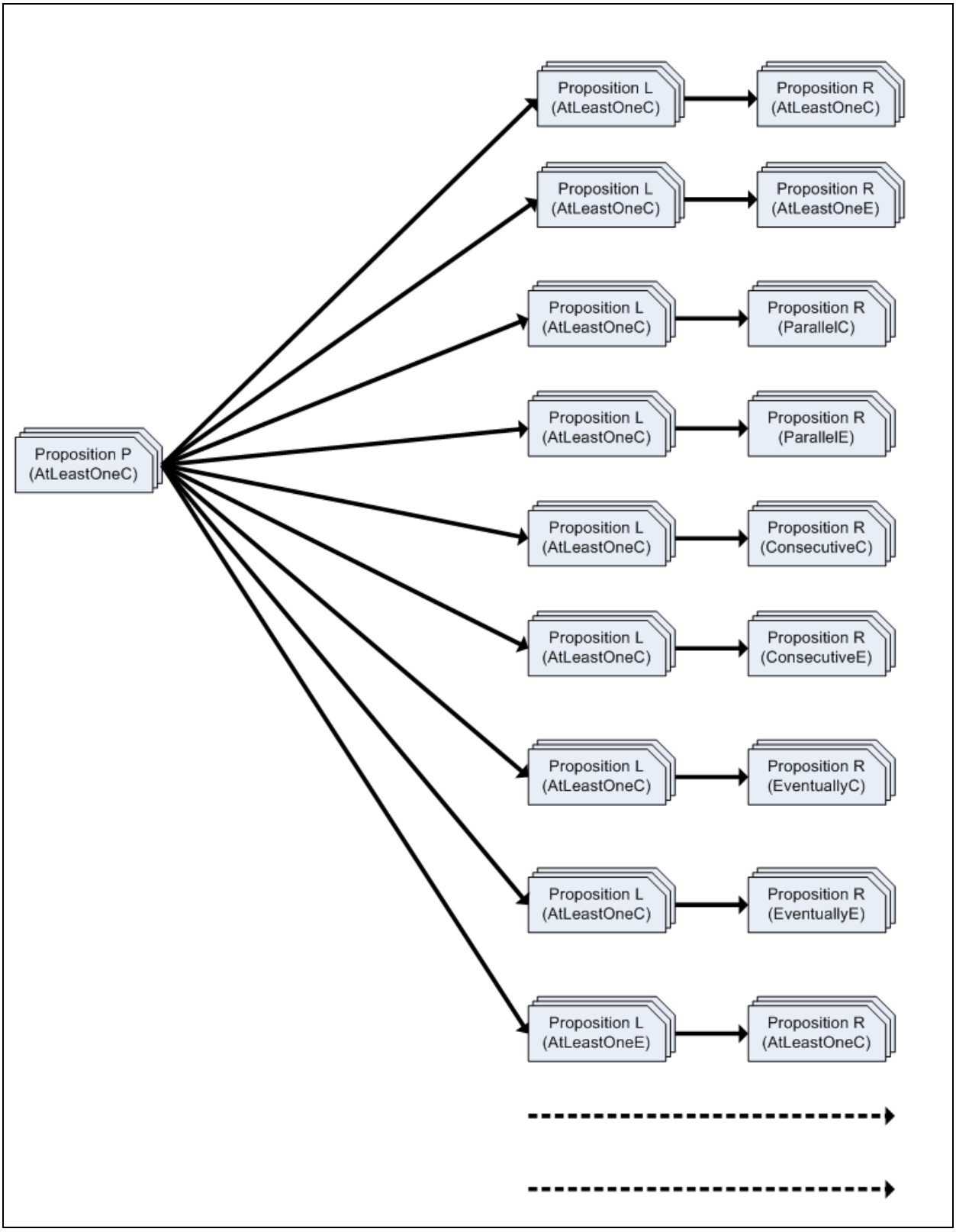


Figure 12: *Absence of P After L until R* combinations.

Table 11: Total Number of CP Combinations.

| | Global | Before R | After L | After L until R | Between L and R |
|---|---|---|---|---|---|
| Absence | 8 | 64 | 64 | 512 | 512 |
| Existence | 8 | 64 | 64 | 512 | 512 |
| Response | 64 | 512 | 512 | 4096 | 4096 |
| Precedence | 64 | 512 | 512 | 4096 | 4096 |
| Strict Precedence | 64 | 512 | 512 | 4096 | 4096 |

### 3.3 Generating Concrete Execution Traces

Generating concrete execution traces require the interaction of two processes: *Apply CP Rules* and *Extract Beginning and Ending States* processes. Applying CP Rules means that each abstract proposition in an abstract execution trace can be defined with one of the eight CP classes (*Set of Abstract Execution Traces* input value for the *Apply CP Rules* process in Figure 10). Once each abstract proposition is defined with a specific CP class, a beginning and ending state (*Extract Beginning* and *Ending State* Process in Figure 10) can be assigned to each abstract proposition in an abstract execution trace (previously stored in the Data Store component in Figure 10). The result of this process is divided into two outputs: a concrete execution trace (or simply execution trace) containing concrete propositions (or simply propositions) indicated by an arrow from Apply CP Rules process to *Construct Oracle* process shown in Figure 10.

The example in Figure 13 shows the pattern *Absence of P (AtLeastOneC) Before R (AtLeastOneC)* combination. Every proposition in the execution traces, shown in Figure 13, is defined with a specific CP, and with ending and beginning states based on the specified CP.

**Absence of P (AtLeastOne$_C$) Before R (AtLeastOne$_C$)**

**Some sample abstract execution traces:**

**P1 R1** ___ ___ ___ ___ ___
*0 1 2 3 4 5 6*

**Beginning States P = {0}**
**Ending States P = {0}**

**Beginning States R = {1}**
**Ending States R = {1}**

**___ ___ ___ ___ R3 P1 R1**
*0 1 2 3 4 5 6*

**Beginning States P = {5}**
**Ending States P = {5}**

**Beginning States R = {4}**
**Ending States R = {4}**

**___ R1 P1 P2 ___ ___ ___**
*0 1 2 3 4 5 6*

**Beginning States P = {2}**
**Ending States P = {2}**

**Beginning States R = {1}**
**Ending States R = {1}**

Figure 13: Concrete Execution Traces Examples

### 3.4 Generating Expected Results

Expected results are obtained by applying semantic rules from SPS to every concrete execution trace. Semantic rules are derived from scope rules and state selections. The result of applying semantic rules is an expected result associated with every concrete execution trace for a given LTL specification, either *satisfied* or *falsified.* For example, the first execution trace in Figure 14 is falsified since it is not the case that P1 is absent before R1.

For a given specification derived from a pattern, scope, and set of CPs, concrete execution traces and expected results comprise the set of *test cases*. For example, test cases in Figure 14 are derived from the pattern, scope, and CPs. The resulting set of test cases is formed by LTL formula specifications, combinations of pattern, scopes, and CPs, and expected and actual results. Figure 15

and Figure 16 show sample test case selection strategies by applying equivalence classes and boundary values analyses.

**Absence of P ($AtLeastOne_C$) Before R ($AtLeastOne_C$)**

**Some sample abstract execution traces:**

**P1 R1 ___ ___ ___ ___ ___** ***FALSIFIED***
*0 1 2 3 4 5 6*

**Beginning States P = {0}**
**Ending States P = {0}**

**Beginning States R = {1}**
**Ending States R = {1}**

**___ ___ ___ ___ R3 P1 R1** ***SATISFIED***
*0 1 2 3 4 5 6*

**Beginning States P = {5}**
**Ending States P = {5}**

**Beginning States R = {4}**
**Ending States R = {4}**

**___ R1 P1 P2 ___ ___ ___** ***SATISFIED***
*0 1 2 3 4 5 6*

**Beginning States P = {2}**
**Ending States P = {2}**

**Beginning States R = {1}**
**Ending States R = {1}**

Figure 14: Test Cases for *Absence of P Before R.*

**Absence of P1 ($AtLeastOne_C$) Global:**

***Valid Equivalence Class:***

___ ___ ___ ___ ___ ___ ___ ___

*As long as P1 does not hold in the execution trace (at any state), the pattern and scope are satisfied by the execution trace.*

***Invalid Equivalence Class:***

___ ___ P1 ___ ___ ___ ___ ___

*As long as one P1 holds in the execution trace (at any state), the pattern and scope are not satisfied by the execution trace.*

**Absence of P1 ($AtLeastOne_C$) Before R1 ($AtLeastOne_C$):**

***Valid Equivalence Classes:***

___ ___ ___ R1 ___ ___ ___ ___

*As long as P1 does not hold in the execution trace before R1 (at any state before R1), the pattern and scope are built in the execution trace.*

***Invalid Equivalence Class:***

___ ___ P1 R1 ___ ___ ___ ___

*As long as P1 holds in the execution trace before R1 (at any state before R1), the pattern and scope are satisfied by the execution trace.*

**Absence of P1 ($AtLeastOne_C$) After L1 ($AtLeastOne_C$):**

***Valid Equivalence Classes:***

___ ___ ___ L1___ ___ ___ ___

*As long as P1 does not hold in the execution trace after L1 (at any state after L1), the pattern and scope are satisfied the execution trace.*

***Invalid Equivalence Classes:***

___ ___ ___ L1 P1 ___ ___ ___

___ ___ ___ (L1P1) ___ ___ ___

*As long as P1 holds in the execution trace after L1 (at any state after L1), or at the same state, the pattern and scope are not satisfied in the execution trace.*

**Absence of P1 ($AtLeastOne_C$) Between L1 ($AtLeastOne_C$) and R1 ($AtLeastOne_C$):**

***Valid Equivalence Classes:***

___ ___ ___ L1 ___ R1 ___ ___

*As long as P1 does not hold in the execution trace between L1 and R1 (at any state between L1 and R1), the pattern and scope are satisfied by the execution trace.*

___ ___ ___ L1 ___ ___ ___

*If R1 is not present in the execution trace, the scope is not built and any occurrence of P1 that may occur (at any state) will not violate the pattern and scope definition.*

***Invalid Equivalence Classes:***
___ ___ ___ L1 P1 R1 ___ ___

*As long as P1 holds in the execution trace between L1 and R1 (at any state between L1 and R1), the pattern and scope are not satisfied by the execution trace.*

___ ___ (P1L1) ___ R1 ___ ___

*If the scope is built, and if P1 and L1 hold in the same state, the pattern and scope are not satisfied by the execution trace.*

**Absence of P1 ($AtLeastOne_C$) After L1 ($AtLeastOne_C$) until R1 ($AtLeastOne_C$):**

***Valid Equivalence Classes:***
___ ___ ___ L1 ___ R1 ___ ___

*As long as P1 does not hold in the execution trace between L1 and R1 (at any state between L1 and R1), the pattern and scope are satisfied by the execution trace.*

___ ___ ___ L1 ___ ___ ___

*If P1 is not present in the execution trace after L1, the pattern and scope are satisfied by the execution trace.*

***Invalid Equivalence Classes:***
___ ___ ___ L1 P1 R1 ___ ___

*As long as P1 holds in the execution trace between L1 and R1 (at any state between L1 and R1), the pattern and scope are not satisfied by the execution trace.*

___ ___ (P1L1) ___ R1 ___ ___

*If the scope is built, and if P1 and L hold in the same state, the pattern and scope are not satisfied by the execution trace.*

___ ___ L1 ___ P1 ___ ___

*If P1 holds after L1, the pattern and scope are not satisfied by the execution trace.*

Figure 15: Test Case selection strategies with equivalence classes.

**Absence of P1 (AtLeastOne$_E$) Global:**

***Valid Equivalence Class:***

___ ___ ___ ___ ___ ___ ___ ___

***Boundary Values:***
*The execution trace is empty, thus no need to check for P1 at the boundaries.*

***Invalid Equivalence Class:***
___ ___ P1 ___ ___ ___ ___ ___

***Boundary Values:***
P1 ___ ___ ___ ___ ____ ___
*Boundary at the start of the execution trace. P1 is not present therefore the CP is not built[1]. Note how an invalid equivalence class with the inclusion of boundary values makes the execution trace satisfy the pattern and scope.*

___ ___ P1 ___ ___ ___ ___
*Boundary at an arbitrary state of the execution trace. P1 is present therefore the pattern and scope are not satisfied by the execution trace.*

___ ___ ___ ___ ___ ___ P1
*Boundary at the end of the execution trace. P1 is present therefore the pattern and scope are not satisfied by the execution trace.*

**Absence of P1 (AtLeastOne$_E$) Before R1 (AtLeastOne$_E$):**

***Valid Equivalence Classes:***
___ ___ ___ R1 ___ ___ ___ ___

[1] P1 is a CP of type *AtLeastOne$_E$*. This means that the beginning and the ending states of P1 must be inside the scope. Clearly, the beginning state of P1 is not present in the execution trace, and thus not present inside the scope.

***Boundary Values:***

R1 ___ ___ ___ ___ ___

*Boundary at the start of the execution trace. R1 is not present in the execution trace and the scope is not built. Thus, the execution trace satisfies the pattern and scope.*

___ ___ R1 ___ ___ ___

*Boundary at an arbitrary state of the execution trace. R1 is present and no Ps are available before R1. Thus, the execution trace satisfies the pattern and scope.*

___ ___ ___ ___ ___ ___ R1

*Boundary at the end of the execution trace. R1 is present and no Ps are available before R1. Thus, the execution trace satisfies the pattern and scope.*

***Invalid Equivalence Class:***

___ ___ P1__ R1 ___ ___ ___ ___

***Boundary Values:***

P1__R1 ___ ___ ___ ___ ___

*Boundary at the start of the execution trace. P1 is not present in the execution trace. P1 is absent before R1, and thus the execution trace satisfies the pattern and scope.*

___ ___ P1__R1 ___ ___ ___

*Boundary at an arbitrary state of the execution trace. P1 is present before R1. Thus the execution trace does not satisfy the pattern and scope.*

___ ___ ___ ___ ___ P1__R1

*Boundary at the end of the execution trace. P1 is present before R1. Thus the execution trace does not satisfy the pattern and scope.*

**Absence of P1 ($AtLeastOne_E$) After L1 ($AtLeastOne_E$):**

Valid Equivalence Classes:

___ ___ ___ L1___ ___ ___ ___

***Boundary Values:***

L1 ___ ___ ___ ___ ___ ___

*Boundary at the start of the execution trace. L1 is not present in the execution trace. The scope is not built, therefore the execution trace satisfies the pattern and scope.*

___ ___ L1 ___ ___ ___

*Boundary at an arbitrary state of the execution trace. P1 is not present after L1. Thus the execution trace satisfies the pattern and scope.*

___ ___ ___ ___ ___ ___ L1
*Boundary at the end of the execution trace. P1 is not present after L1. Thus the execution trace satisfies the pattern and scope.*

Invalid Equivalence Classes:
___ ___ ___ L1 P1 ___ ___ ___

L1P1 ___ ___ ___ ___ ___
*Boundary at the start of the execution trace. L1 is not present in the execution trace. The scope is not built. Therefore, the execution trace satisfies the pattern and scope.*

___ ___ L1P1 ___ ___ ___
*Boundary at an arbitrary state of the execution trace. P1 is present after L1. Thus the execution trace does not satisfy the pattern and scope.*

___ ___ ___ ___ ___ L1P1
*Boundary at the end of the execution trace. P1 is present after L1. Thus the execution trace does not satisfy the pattern and scope.*

___ ___ ___ (L1P1) ___ ___ ___

(L1P1) ___ ___ ___ ___ ___
*Boundary at the start of the execution trace. L1 and P1 are not present in the execution trace. The scope is not built. Therefore, the execution trace satisfies the pattern and scope.*

___ ___ (L1P1) ___ ___ ___
*Boundary at an arbitrary state of the execution trace. P1 is present after L1. Thus the execution trace does not satisfy the pattern and scope.*

___ ___ ___ ___ ___ (L1P1)
*Boundary at the end of the execution trace. P1 is present after L1. Thus the execution trace does not satisfy the pattern and scope.*

Figure 16: Test Case selection strategies with boundary values analyses.

### 3.5 Selecting States

The process for selecting states requires the semantics of patterns as input values. Based on the pattern semantics, the *Select State* process, shown in Figure 10, produces an output value in the form of state sets. *States* represent the set of boundary values that can be represented in the execution traces. There are three types of states that can be applied to SPS and CPs: *Start State, Middle State, and End State. Start State* defines the set of boundary conditions at the beginning of an execution

trace. *Middle State* defines the set of boundary conditions at arbitrary representative state between the Start and End states. Finally, *End State* defines the set of boundary conditions at the end of an execution trace. Figure 17 shows *Absence of P* with its corresponding states.

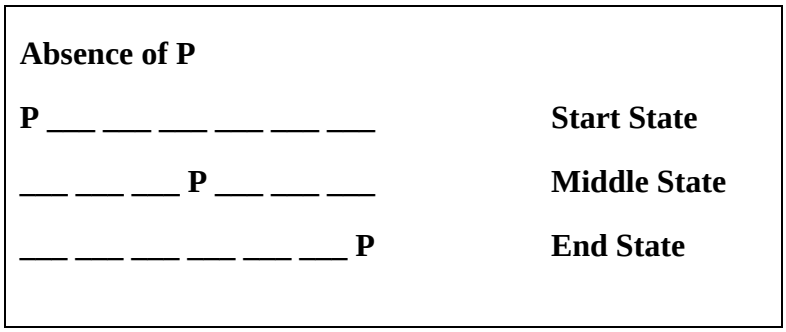


Figure 17: Abstract Execution Traces for States for *Absence of P*.

The examples in Figure 17 are templates indicating the locations of possible propositions. These examples do not represent actual execution traces, because they do not contain information about patterns, scopes, and CPs. Instead, they represent *abstract execution traces* that can have several meanings based on patterns, scopes, and CPs (abstract execution traces explained in detail in section 3.3). Possible propositions, represented by Ps in Figure 17, are placeholders for the actual propositions. Actual propositions come from pattern, scope, and CP combinations (e.g. *Absence of P* (*P1*, *P2*, and *P3*) *Before R* (*R1*, *R2*, and *R3*)), and these actual propositions replace the placeholders (explained in section 3.3).

Although the selection of states generates abstract execution traces with placeholders, there is no means to identify the extent to which possible propositions hold in the abstract execution traces. Therefore, it is necessary to identify the scope of possible propositions, and apply rule mechanisms that can incorporate scopes to the abstract execution traces.

### 3.6 Selecting Scope Rules

The application of scopes to abstract execution traces comes in the form of two rules: the P-Rule and the PR-Rule. The *P-Rule* describes a set of rules where only one possible proposition or

placeholder is present. For example, *Existence* and *Absence* patterns combined with the *Global* scope require only one proposition. The only proposition required is the one provided by the pattern (e.g. *Absence of P Global*).The *Global* scope does not take any proposition, and therefore the only proposition considered is the one provided by the pattern. Also, since only one proposition is needed, *Precedence*, *Strict Precedence*, and *Response* do not fall under this rule as they require two propositions. Single proposition means that if we have *Absence of P*, *P* can be replaced by *P1*, *P2*, or *P3*. Figure 18 shows an example of P-Rule with *Absence of P Before R*.

**Absence of P Before R**

**Applying P-Rule to P and R:**

**Input:** P-Rule(P)
**Output:** A set containing **{**P1, P2, P3, (P1P2), (P1P3), (P2P3)}

**Input:** P-Rule(R)
**Output:** A set containing **{**R1, R2, R3, (R1R2), (R1R3), (R2R3)}

Figure 18: P-Rule with *Before R*.

The P-Rule only defines single propositions, which are not very useful when combining patterns with scopes such as *Before* R and *After* L. That is, we are interested in verifying that some P occurs before some R, and not whether a single P occurs. *PR-Rule* defines a combination of more than one proposition. Figure 19 shows PR-Rule applied to *Absence of P Before R*.

**Absence of P Before R**

**Applying PR-Rule to P and R:**

**Input:** PR-Rule(P, R)
**Output:{** P1R1, P1R2, P1R3, P2R1, P2R2, P2R3, P3R1, P3R2,…., (P1R1), (P1R2), (P1R3), (P2R1), (P2R2), (P2R3), (P3R1), (P3R2),…, P1--R1, P1--R2, P1--R3, P2--R1, P2--R2, P2--R3, P3--R1, P2--R2,…, R1P1, R1P2, R1P3, R2P1, R2P2, R2P3, R3P1, R3P2,…, P1P2P3, P1P2P3R1, P1P2P3R2,……..,---}

Figure 19: PR-Rule with *Absence of P Before R*.

The resulting output in Figure 19 includes the combination of Ps and Rs. This is useful when verifying that *P1* occurs before *R1*.

The example shown in Figure 20 demonstrates how the test cases for *Absence of P ($AtLeastOne_C$) Before R ($AtLeastOne_E$)* are generated by applying all the steps explained before.

Objective: To construct test cases for *Absence of P ($AtLeastOne_C$) Before R ($AtLeastOne_E$)* where P = P1, P2, and P3, and R = R1, R2, and R3.

**Step 1:** Select States for the pattern. At this point, we are considering the pattern definition only, thus P is the placeholder being used. Note that the proposition P is abstract, meaning that it can be any proposition (i.e. R, L, M, etc.).

| | |
|---|---|
| P___ ___ ___ ___ ___ | Start State |
| ___ ___ ___ P___ ___ | Middle State |
| ___ ___ ___ ___ ___ P | End State |

**Step 2:** Define the propositions for pattern and scope that will eventually lead to composite propositions. These represent the possible combinations for P, R, and P and R together.

P-Rule(P): {P1, P2, P3, (P1P2), (P1P3), (P2P3), ---}

P-Rule(R): {R1, R2, R3, (R1R2), (R1R3), (R2R3), ---}

PR-Rule(P,R): **{**P1R1, P1R2, P1R3, P2R1, P2R2, P2R3, P3R1, P3R2,…., (P1R1), (P1R2), (P1R3), (P2R1), (P2R2), (P2R3), (P3R1), (P3R2),…, P1--R1, P1--R2, P1--R3, P2--R1, P2--R2, P2--R3, P3--R1, P2--R2,…, R1P1, R1P2, R1P3, R2P1, R2P2, R2P3, R3P1, R3P2,…, P1P2P3, P1P2P3R1, P1P2P3R2,……..,---}

*Note that "---" means the empty trace which is included in every rule

**Step 3:** Define abstract execution traces (i.e. execution traces with no CP definition).

By substituting each rule in the States from step 1 we get:

For P-Rule(P) (P-Rule(R) is similar to P-Rule(P)):

Propositions for P-Rule were selected as follows:

P1 = Selected from P-Rule(P) = {P1,…}

P2 = Selected from P-Rule(P) = {.P2,…}

P3 = Selected from P-Rule(P) = {..P3..}

(P1P2) = Selected from P-Rule(P) = {….(P1P2)}

(P2P3) = Selected from P-Rule(P) = {.(P2P3…}

----- = Selected from P-Rule(P) = {….-----}

**Start State**

P1 ___ ___ ___ ___ ___

P2 ___ ___ ___ ___ ___

P3 ___ ___ ___ ___ ___

(P1P2) ___ ___ ___ ___

(P2P3) ___ ___ ___ __

____ ___ ___ ___ ___ ___

…….

**Middle State**

___ ___ P1 ___ ___ ___

___ ___ P2 ___ ___ ___

___ ___ P3 ___ ___ ___

___ ___ (P1P2) ___ ___

___ ___ (P2P3) ___ ___

……

**End State**

___ ___ ___ ___ ___ P1

___ ___ ___ ___ ___ P2

___ ___ ___ ___ ___ P3

___ ___ ___ ___ ___ (P1P2)

___ ___ ___ ___ ___ (P2P3)

……

For PR-Rule(P, R):

Propositions for PR-Rule were selected as follows:

P1R1 = Selected from PR-Rule(P) = {P1R1,…}

(P1R1) = Selected from PR-Rule(P) = {.(P1R1),,…}

P1----R2 = Selected from PR-Rule(P) = {..P1--R2..}

----- = Selected from PR-Rule(P,R) = {….-----}

**Start State**

P1R1 ___ ___ ___ ___

(P1R1) ___ ___ ___ ___

P1 ___ ___ R2 ___ ___

___ ___ ___ ___ ___ ___

……..

**Middle State**

___ ___ P1R1 ___ ___

___ ___ (P1R1) ___ ___

___ ___ P1 ___ ___ R2 ___ ___

……..

**End State**

___ ___ ___ ___ P1R1

___ ___ ___ ___ (P1R1)

___ ___ ___ ___ P1 ___ ___ R2

……..

**Step 4:** Define Composite Propositions by defining beginning and ending states. Concrete execution traces (CP definitions) are defined here. Some execution traces are shown below.

P1 ___ ___ ___ ___ ___
0 1 2 3 4 5
Beginning State (P) = {0}
Ending State (P) = {0}

P2 ___ ___ ___ ___ ___
0 1 2 3 4 5
Beginning State (P) = {0}
Ending State (P) = {0}

(P1R1) ___ ___ ___ ___
0 1 2 3 4
Beginning State (P) = {0}
Beginning State (R) = {undefined}
Ending State (P) = {0}
Ending State (R) = {0}

P1 ___ ___ R2 ___ ___
0 1 2 3 4 5
Beginning State (P) = {0}
Beginning State (R) = {2}
Ending State (P) = {0}
Ending State (R) = {3}
…….

**Step 5:** Provide expected results by analyzing each concrete execution trace.

P1 ___ ___ ___ ___ ___ SATISFIED

0 1 2 3 4 5

Beginning State (P) = {0}

Ending State (P) = {0}

Checking if: Beginning State(P) {0} and Ending State(P) {0} >= Ending State(R) {undefined}

Since the beginning and ending state of R is not present, the scope is not built and thus the execution trace satisfies the pattern, scope, and CP specification.

P2 ___ ___ ___ ___ ___ SATISFIED

0 1 2 3 4 5

Beginning State (P) = {0}

Ending State (P) = {0}

Checking if: Beginning State(P) {0} and Ending State(P) {0} >= Ending State(R) {undefined}

Since the beginning and ending states of R are not present, the scope is not built and thus the execution trace satisfies the pattern, scope, and CP specification.

(P1R1) ___ ___ ___ ___ SATISFIED

0 1 2 3 4

Beginning State (P) = {0}

Beginning State (R) = {undefined}

Ending State (P) = {0}

Ending State (R) = {0}

Checking if: Beginning State(P) {0} and Ending State(P) {0}>= Ending State(R) {0} but Beginning State(R) {undefined}

Since the beginning state of R is undefined (not present), the scope is not built and thus the execution trace satisfies the pattern, scope, and CP specification.

P1 ___ ___ R2 ___ ___ FALSIFIED

0 1 2 3 4 5

Beginning State (P) = {0}

Beginning State (R) = {2}

Ending State (P) = {0}

Ending State (R) = {3}

Checking if: Beginning State(P) {0} <= Beginning State (R) {0} and Ending State (or Beginning State) (P) {0} < Ending State (R) {3}.

Since the beginning and ending states of R is present, and P (P1) is present before R (R2 at state 3), the scope is built and P (P1) is not absent before R (R2).

…….

Figure 20: Test Case Example.

## Chapter 4: Implementation of the LTL Verifier

The algorithm presented in Chapter 3 was implemented in Java to test Prospec's LTL Generator. The input values to the LTL Verifier are the semantics of patterns, scopes, CPs, and the output values are set of execution traces with expected results. The process proceeds as follows.

1. Input values are provided by the user in plain text. This includes the specification of patterns, pattern's propositions, CPs for pattern's proposition, scopes, scope's propositions, and CPs for scope's propositions.
2. The input data provided by the user in plain text is passed to the LTL Generator to generate LTL formulas.
3. The LTL Verifier translates LTL formulas' operators (i.e. AND, OR, or NOT specified in text in the LTL Generator) into NuSMV's LTL operators (e.g. AND = &).
4. Based on patterns, scopes, and CPs, the LTL Verifier automatically generates abstract execution traces by applying state selection and scope rules processes.
5. The LTL verifier applies CP rules, and extracts beginning and ending states from CPs to generate concrete execution traces. Expected results are automatically obtained by analyzing concrete execution traces and their respective propositions.

With the application of these six steps (steps explained in detail in section 4.1), test cases are automatically generated. Test cases are composed of LTL formulas, execution traces, and the expected and actual results. The actual results are obtained by running PROTEF. The process for automatically generating more than 3,800,000 test cases is explained in the last section of Chapter 4 (Section 4.2).

## 4.1 Implementation Process

First, users will be responsible for providing patterns, scopes, CPs, and propositions (in plain text) as shown in Figure 21. The text file or input file will serve as a script for generating LTL formulas and execution traces automatically. In general, we think of a specification as consisting of an abstract execution trace refined by composite propositions. The specification has a left and right

boundary that define the scope, denoted by L and R respectively. The patterns may contain zero, one, or two markers, denoted by P and T. For example, we can specify that for scope *between L and R* and pattern *P responds to T*, the abstract trace will have "LTPR". Not all markers are used in every pattern and scope. Each marker is refined by a composite proposition. For the purpose of testing, each CP may have one, two, or three actual propositions, which are labeled with a numbered marker name, e.g., R1, R2, and R3 for abstract marker R. CPs must be one of the following: $AtLeastOne_C$, $AtLeastOne_E$, $Parallel_C$, $Parallel_E$, $Consecutive_C$, $Consecutive_E$, $Eventually_C$, or $Eventually_E$. The description for each label in the input file is as follow:

- **Pattern**: One of the following patterns: *Absence, Existence, Precedence, Strict Precedence,* or *Response.*
- **Composite Proposition for P, T, R, and L**: One of the following: $AtLeastOne_C$, $AtLeastOne_E$, $Parallel_C$, $Parallel_E$, $Consecutive_C$, $Consecutive_E$, $Eventually_C$, or $Eventually_E$.
- **Propositions P**: At least one P1, or at most P1, P2, and P3.
- **Propositions T**: At least one T1, or at most T1, T2, and T3 (if *Response*, *Precedence*, or *Strict Precedence* is present, "none" must be specified).
- **Scope**: One of the following: *Global, Before R, After L, Between L and R,* or *After L until R.*
- **Propositions R**: At least R1 and at most R1, R2, and R3 (if *Before R,* or *Between L and R,* or *After L until R* is specified as the pattern, otherwise this label must have "none").
- **Propositions L**: At least L1 and at most L1, L2, and L3 (if *After L* or, *After L until R,* or *Between L and R* is specified as the pattern, otherwise this label must have "none").

**Pattern**: Absence
**Propositions P:** P1, P2, P3
**Composite Proposition for P:** $\text{AtLeastOne}_E$
**Propositions T**: none.
**Composite Proposition for T:** none
**Scope**: Before R
**Propositions R**: R1, R2, R3
**Composite Proposition for R:** $\text{AtLeastOne}_E$
**Propositions L:** none
**Composite Proposition for L:** none

Figure 21: Sample input from the user in plain text

Second, the LTL Generator generates LTL formulas from patterns, scopes, and CPs [Salamah 2007]. The specifications for generating LTL formulas are provided in the input file. Third, The LTL Verifier obtains the LTL formulas and formats them for use by the NuSMV model checker. For example, instead of having (P1 AND P2) OR (P1 AND P2), the new formula must be (P1 & P2) | (P1 & P2). For example, let the pattern be *Absence of P* with the scope *Before R*, and let the CP be $AtLeastOne_E$ for both pattern and scope. The resulting LTL formula (after translation) is shown in Figure 22.

```
((F  (( !R1 & !R2 & !R3) & (( !R1 & !R2 & !R3) U (R1 | R2 |
R3)))) -> !((!(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) U (((!P1 &
!P2 & !P3) & !(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) & (((!P1 &
!P2 & !P3) & !(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) U ((P1 |
P2 | P3) & !(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 )))))))
```

Figure 22: LTL formula: *Absence of P Before R* with $AtLeastOne_E$.

Fourth, the LTL Verifier automatically generates abstract execution traces by applying selection states and scope rules. Fifth, the LTL Verifier automatically generates concrete execution traces by applying CP rules, and by extracting beginning and ending states from CPs. Sixth, expected results are obtained by analyzing each proposition in the execution traces. Figure 23 shows how expected results are obtained with one execution trace example. Figure 24 shows the resulting

set of execution traces with expected results for *Absence of P (AtLeastOne$_E$) Before R (AtLeastOne$_E$)*.

***Absence of P (AtLeastOne$_E$) Before R (AtLeastOne$_E$)***

**Some sample execution traces:**

**P1** ___ ___ ___ ___ **SATISFIED**
*0 1 2 3 4*

```
if(propositions for pattern present){
    if(scope built){
        if(ending and beginning states of proposition P
            using AtLeastOne_E are before R){
            FALSIFIED;
        }
        else{
         SATISFIED;
        }
    }
    else{
       SATISFIED;
    }
}
else{
  SATISFIED;
}
```

Figure 23: LTL formula: *Absence of P Before R* with *AtLeastOne$_E$*.

```
Pattern: Absence
Scope: Before R
CP Class: AtLeastOneE(P)
CP Class: AtLeastOneE(R)
P1------------------              SATISFIED
P2------------------              SATISFIED
P3------------------              SATISFIED
R1------------------              SATISFIED
R2------------------              SATISFIED
R3------------------              SATISFIED
P1P2P3R1R2R3------------------            SATISFIED
R3R2R1P3P2P1------------------            SATISFIED
(P1P2P3R1R2R3)----------------            SATISFIED
P1-P2-P3-R1-R2-R3-------------------      FALSIFIED
(P1)P2P3R1R2R3----------------            SATISFIED
(P1P2)P3R1R2R3----------------            SATISFIED
P1R1R2R3----------------                  SATISFIED
P1P2R1R2R3----------------                SATISFIED
```

```
R1P1P2P3------------------                SATISFIED
R1R2P1P2P3----------------------          SATISFIED
R3--------------------              SATISFIED
----------P1---------              SATISFIED
----------P2---------              SATISFIED
----------P3---------              SATISFIED
----------R1---------              SATISFIED
----------R2---------              SATISFIED
----------R3---------              SATISFIED
----------P1P2P3R1R2R3---------            FALSIFIED
----------R3R2R1P3P2P1---------            SATISFIED
----------(P1P2P3R1R2R3)---------          SATISFIED
----------P1-P2-P3-R1-R2-R3----------      FALSIFIED
----------(P1)P2P3R1R2R3---------          FALSIFIED
----------(P1P2)P3R1R2R3---------          FALSIFIED
----------(P1P2P3)R1R2R3---------          SATISFIED
----------P2P1P3R1R2R3---------            FALSIFIED
----------P3P1P2R1R2R3---------            FALSIFIED
----------R1P1P2P3R2R3---------            SATISFIED
----------R2P1P2P3R1R3---------            SATISFIED
----------R3P1P2P3R1R2---------            SATISFIED
----------P1R3R2R1-P3P2---------           SATISFIED
----------P2R3R2R1-P3P1----------          SATISFIED
----------R3R2R1-P3P2P1----------          SATISFIED
----------R3----------             SATISFIED
------------------P1-              SATISFIED
------------------P2-              SATISFIED
------------------P3-              SATISFIED
------------------R1-              SATISFIED
------------------R2-              SATISFIED
------------------R3-              SATISFIED
------------------P1P2P3R1R2R3-            FALSIFIED
------------------R3R2R1P3P2P1-            SATISFIED
------------------(P1P2P3R1R2R3)-          SATISFIED
------------------R3--             SATISFIED
--------------------               SATISFIED
```

Figure 24: Execution traces for *Absence of P Before R* with $AtLeastOne_E$.

Next, each execution trace and the LTL formula are passed to PROTEF, which generates NuSMV models. A model file containing the LTL formula and the model constructed from the execution trace is generated for each execution trace. A sample model generated from PROTEF, the execution trace, and the expected result are shown in Figure 25 (the example in Figure 25 represents a test case). The definition of each label in the model file (Figure 25) is as follows:

- **Header Label:**
    - **Passed:** This can be either "true" or "false." It is true if a test case passes, otherwise it is false.

- **Expected:** The expected result as specified in the execution traces in Figure 24 (e.g. SATISFIED = true, FALSIFIED = false).

- **Original Execution Trace:** The execution trace and the expected result generated from the LTL Verifier.

- **Model File:** Model definition from NuSMV (automatically generated by PROTEF).
  - **LTLSPEC:** LTL formula generated by the LTL Generator.
  - **NuSMV source code:** Specifies the states used by propositions in the execution trace.

```
Passed : true
Expected : true but was : true

---- Original Execution Trace ----

(P1P2P3R1R2R3)------------------         SATISFIED

---- Model File ----
MODULE main
VAR
  Q : seq();
  DEFINE
         P1 := (Q.State = 0);
         P2 := (Q.State = 0);
         P3 := (Q.State = 0);
         R1 := (Q.State = 0);
         R2 := (Q.State = 0);
         R3 := (Q.State = 0);

LTLSPEC  ((F  (( !R1 & !R2 & !R3) & (( !R1 & !R2 & !R3) U (R1 | R2
| R3)))) -> !((!(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) U (((!P1 &
!P2 & !P3) & !(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) & (((!P1 &
!P2 & !P3) & !(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) U ((P1 | P2 |
P3) & !(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 )))))))


MODULE seq()
VAR
        State: 0..19;2
ASSIGN
        init(State):= 0;
        next(State):= case
                (State != 19) : {State + 1};
                (State = 19)  : {19};
                esac;
```

Figure 25: PROTEF model for *Absence of P Before R* with $AtLeastOne_E$.

### 4.2 Total Number of Test Cases

The total number of test cases obtained from the implementation of the LTL Verifier was 3,836,960. Table 12 shows the number of test cases for each pattern and scope. The number of test cases for each pattern is obtained by multiplying the number of execution traces times the number of combinations. For example, as shown in Figure 18 (*Absence of P after L until R*), 512 combinations are possible. We use 106 test cases for just this combination. Therefore, we obtain 54,272 test cases.

Table 12: Number of test cases for each pattern and scope.

| **Pattern** | **Global** | **Before R** | **After L** | **After L until R** | **Between L and R** |
|---|---|---|---|---|---|
| Absence | 208 | 5056 | 5056 | 54,272 | 54,272 |
| Existence | 208 | 5056 | 5056 | 54,272 | 54,272 |
| Response | 1,664 | 54,272 | 54,272 | 544,768 | 544,768 |
| Precedence | 1,664 | 54,272 | 54,272 | 544,768 | 544,768 |
| Strict Precedence | 1,664 | 54,272 | 54,272 | 544,768 | 544,768 |

## Chapter 5: Results of Testing Prospec's LTL Generator

The number of test cases generated through the LTL Verifier was 3,836,960 of which 79,958 test cases (2%) failed. All of these test cases were executed. Figure 39 shows a graph with the number of test cases that failed for each pattern. The errors were classified according to the error types described below.

### 5.1 Analysis of Errors in the LTL Generator

After running test cases for all patterns, scopes, and CPs supported by the LTL Generator, we discovered problems in the LTL formulas generated. All the errors were found in the implementation of the LTL Generator. Three types of errors were identified:

1. Missing parentheses when negated propositions are followed by the *until* operator.
2. Missing open and close parentheses.
3. More than one missing open parenthesis.

Errors of type 1 were identified only when the proposition L was of type $Eventually_E$. Errors of type 2 and type 3 were discovered across every pattern. Figure 26 shows a test case that failed for the *Absence* pattern, *Between L and R* scope with CPs $AtLeastOne_C$ for P, $AtLeastOne_C$ for L, and $AtLeastOne_E$ for R. The part that failed in the LTL formula is also highlighted in Figure 26. The expected result is false but the actual result from the model checker is true. However, the actual result is wrong according to the definitions of SPS and CPs. The scope *Between L and R* holds from L2 to P3. This is because the beginning and ending state for L is where L2 holds, and the beginning state for R is where P3 holds, and the ending state for R is where R1 holds. Thus, P is not absent between L and R. P1 and P2 are clearly between L and R. The correct formula must be as shown in Figure 27. The discovery of this test case was obtained after analyzing a similar test case with different CPs.

```
Passed : false
Expected : false but was : true

---- Original Execution Trace ----

L2P1P2P3R1R2R3L1L3------------------            FALSIFIED

---- Model File ----
MODULE main
VAR
  Q : seq();
  DEFINE
         P1 := (Q.State = 1);
         P2 := (Q.State = 2);
         P3 := (Q.State = 3);
         R1 := (Q.State = 4);
         R2 := (Q.State = 5);
```

```
          R3 := (Q.State = 6);
          L1 := (Q.State = 7);
          L2 := (Q.State = 0);
          L3 := (Q.State = 8);

LTLSPEC  (G (( L1 | L2 | L3 ) & !(!R1 & !R2 & !R3 & X ( R1 | R2 |
R3 ))) -> (( L1 | L2 | L3 ) & ((F  (( !R1 & !R2 & !R3) & (( !R1 &
!R2 & !R3) U (R1 | R2 | R3)))) -> !((!(!R1 & !R2 & !R3 & X ( R1 |
R2 | R3 ))) U ((P1 | P2 | P3) & !(!R1 & !R2 & !R3 & X ( R1 | R2 |
R3 )))))))


MODULE seq()
VAR
        State: 0..27;
ASSIGN
        init(State):= 0;
        next(State):= case
                (State != 27) : {State + 1};
                (State = 27)  : {27};
                esac;
```

Figure 26: *Absence of P Between L and R* test case failure. Place of missing parentheses are highlighted.

```
LTLSPEC  (G ((( L1 | L2 | L3 ) & !(!R1 & !R2 & !R3 & X ( R1 | R2 |
R3 ))) -> (( L1 | L2 | L3 ) & ((F  (( !R1 & !R2 & !R3) & (( !R1 &
!R2 & !R3) U (R1 | R2 | R3))))) -> !((!(!R1 & !R2 & !R3 & X ( R1 |
R2 | R3 ))) U ((P1 | P2 | P3) & !(!R1 & !R2 & !R3 & X ( R1 | R2 |
R3 )))))))
```

Figure 27: Correct LTL formula for *Absence of P Between L and R.*

Another example of a test case that failed is shown in Figure 28. Again, this is problem with missing parentheses in the LTL formula. Clearly, P1, P2, and P3 are true after L3. Thus, the LTL formula is violated as the pattern is *Absence of P (AtLeastOne$_C$)* within the scope *After L (Eventually$_E$) until R (AtLeastOne$_C$)*. Every time there is a group of negated propositions followed by the until operator (e.g. (!L1 & !L2 & !L3), the negated propositions must be enclosed by parenthesis. Figure 29 shows the correct LTL formula.

```
Passed : false
Expected : false but was : true

---- Original Execution Trace ----

-----------------R1R2R3L1L2L3P1P2P3-        FALSIFIED

---- Model File ----
MODULE main
VAR
```

```
  Q : seq();
  DEFINE
      P1 := (Q.State = 24);
      P2 := (Q.State = 25);
      P3 := (Q.State = 26);
      R1 := (Q.State = 18);
      R2 := (Q.State = 19);
      R3 := (Q.State = 20);
      L1 := (Q.State = 21);
      L2 := (Q.State = 22);
      L3 := (Q.State = 23);

LTLSPEC  (G ((((!L1 & !L2 & !L3) & ((!L1 & !L2 & !L3) U (L1 & !L2 &
!L3 & X(!L2 & !L3 U (L2 & !L3 & X(!L3 U (( L3 & (!( R1 | R2 | R3 ))
)))))))) -> ((!L1 & !L2 & !L3) & ((!L1 & !L2 & !L3) U (L1 & !L2 &
!L3 & X(!L2 & !L3 U (L2 & !L3 & X(!L3 U (( L3 & ((!((!( R1 | R2 |
R3 )) U (((P1 | P2 | P3) & (F ( R1 | R2 | R3 ))) & (!( R1 | R2 | R3
))))) & ((!F ( R1 | R2 | R3 )) -> (G !( P1 | P2 | P3 ))))
)))))))))


MODULE seq()
VAR
    State: 0..27;
ASSIGN
    init(State):= 0;
    next(State):= case
        (State != 27) : {State + 1};
        (State = 27)  : {27};
        esac;
```

Figure 28: *Absence of P After L until R* test case failure.

```
LTLSPEC  (G ((((!L1 & !L2 & !L3) & ((!L1 & !L2 & !L3) U (L1 & !L2 &
!L3 & X((!L2 & !L3) U (L2 & !L3 & X(!L3 U (( L3 & (!( R1 | R2 | R3
)) )))))))) -> ((!L1 & !L2 & !L3) & ((!L1 & !L2 & !L3) U (L1 & !L2
& !L3 & X((!L2 & !L3) U (L2 & !L3 & X(!L3 U (( L3 & ((!((!( R1 | R2
| R3 )) U (((P1 | P2 | P3) & (F ( R1 | R2 | R3 ))) & (!( R1 | R2 |
R3 ))))) & ((!F ( R1 | R2 | R3 )) -> (G !( P1 | P2 | P3 ))))
)))))))))
```

Figure 29: Correct LTL formula for *Absence of P After L until R* test case.

All the test cases with the missing parentheses were observed only when the CPs were of type *event*. The templates for CPs of type event contain many parentheses. Figure 30 shows an obvious error with the actual result from the model checker, but it also shows that errors in some LTL formulas are not obvious. The pattern is *T (AtLeastOne$_C$) Strictly Precedes P (AtLeastOne$_C$)* within the *Global* scope.

```
Passed : false
Expected : false but was : true
```

```
---- Original Execution Trace ----

-----------------(P1)P2P3T1T2T3-         FALSIFIED

---- Model File ----
MODULE main
VAR
  Q : seq();
  DEFINE
         T1 := (Q.State = 21);
         T2 := (Q.State = 22);
         T3 := (Q.State = 23);
         P1 := (Q.State = 18);
         P2 := (Q.State = 19);
         P3 := (Q.State = 20);

LTLSPEC  (!((!((!T1 & !T2 & !T3) & !( P1 | P2 | P3 )) & (((!T1 &
!T2 & !T3) & !( P1 | P2 | P3 )) U ((T1 | T2 | T3) & !( P1 | P2 | P3
)))) U ( P1 | P2 | P3 )))


MODULE seq()
VAR
        State: 0..24;
ASSIGN
        init(State):= 0;
        next(State):= case
                (State != 24) : {State + 1};
                (State = 24)  : {24};
                esac;
```

Figure 30: *T Strictly Precedes P Global* test case failure.

The result for the test in Figure 30 must be *falsified*. For every P in the execution trace, T must precede P. In the execution trace, there is no T preceding P1, but the actual result is *satisfied*. This is because the LTL formula is missing some parentheses. The correct LTL formula is shown in Figure 31.

```
(!((!(((!T1 & !T2 & !T3) & !( P1 | P2 | P3 )) & (((!T1 & !T2 & !T3)
& !( P1 | P2 | P3 ))) U ((T1 | T2 | T3) & !( P1 | P2 | P3 )))) U (
P1 | P2 | P3 )))
```

Figure 31: Correct LTL formula for *T Strictly Precedes P Global*.

Finally, the test case in Figure 32 shows a complex LTL formula that is missing several parentheses. This is another error in the implementation of the template for the pattern *T (AtLeastOne$_E$) Precedes P (AtLeastOne$_C$)* within the scope *Between L (AtLeastOne$_E$) and R*

*(AtLeastOne$_E$).* As the combination of patterns, scopes, and CP increases, the LTL formula becomes more complex. The correct LTL formula is shown in Figure 33.

```
Passed : false
Expected : false but was : true

---- Original Execution Trace ----

------------------L1P1P2P3T1T2T3R1R2R3L2L3-
        FALSIFIED

---- Model File ----
MODULE main
VAR
  Q : seq();
  DEFINE
          T1 := (Q.State = 22);
          T2 := (Q.State = 23);
          T3 := (Q.State = 24);
          P1 := (Q.State = 19);
          P2 := (Q.State = 20);
          P3 := (Q.State = 21);
          R1 := (Q.State = 25);
          R2 := (Q.State = 26);
          R3 := (Q.State = 27);
          L1 := (Q.State = 18);
          L2 := (Q.State = 28);
          L3 := (Q.State = 29);

LTLSPEC  (G ( (( !L1 & !L2 & !L3) & (( !L1 & !L2 & !L3) U (L1 | L2
| L3))) & !(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) -> ((!L1 & !L2 &
!L3) & ((!L1 & !L2 & !L3) U (( L1 | L2 | L3 ) & ((F  (( !R1 & !R2 &
!R3) & (( !R1 & !R2 & !R3) U (R1 | R2 | R3)))) -> (!((P1 | P2 | P3)
& !(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) U (((!T1 & !T2 & !T3 &
!( P1 | P2 | P3 )) & ((!T1 & !T2 & !T3 & !( P1 | P2 | P3 )) U (( T1
| T2 | T3 ) & !( P1 | P2 | P3 ))) | (!R1 & !R2 & !R3 & X ( R1 | R2
| R3 ))))))))))


MODULE seq()
VAR
        State: 0..30;
ASSIGN
        init(State):= 0;
        next(State):= case
                (State != 30) : {State + 1};
                (State = 30)  : {30};
                esac;
```

Figure 32: *T Precedes P Between L and R* test case sample.

```
G ((( !L1 & !L2 & !L3) & (( !L1 & !L2 & !L3) U (L1 | L2 | L3))) &
!(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) -> ((!L1 & !L2 & !L3) &
```

```
((!L1 & !L2 & !L3) U (( L1 | L2 | L3 ) & ((F  (( !R1 & !R2 & !R3) &
(( !R1 & !R2 & !R3) U (R1 | R2 | R3)))) -> ((!((P1 | P2 | P3) &
!(!R1 & !R2 & !R3 & X ( R1 | R2 | R3 ))) U (((!T1 & !T2 & !T3 & !(
P1 | P2 | P3 )) & ((!T1 & !T2 & !T3 & !( P1 | P2 | P3 ))))) U ((
T1 | T2 | T3 ) & !( P1 | P2 | P3 )) | (!R1 & !R2 & !R3 & X ( R1 |
R2 | R3 )))))))))
```

Figure 33: Correct LTL formula for *T Precedes P Between L and R*.

Figure 34 shows the total number of passed and failed test cases for each combination.

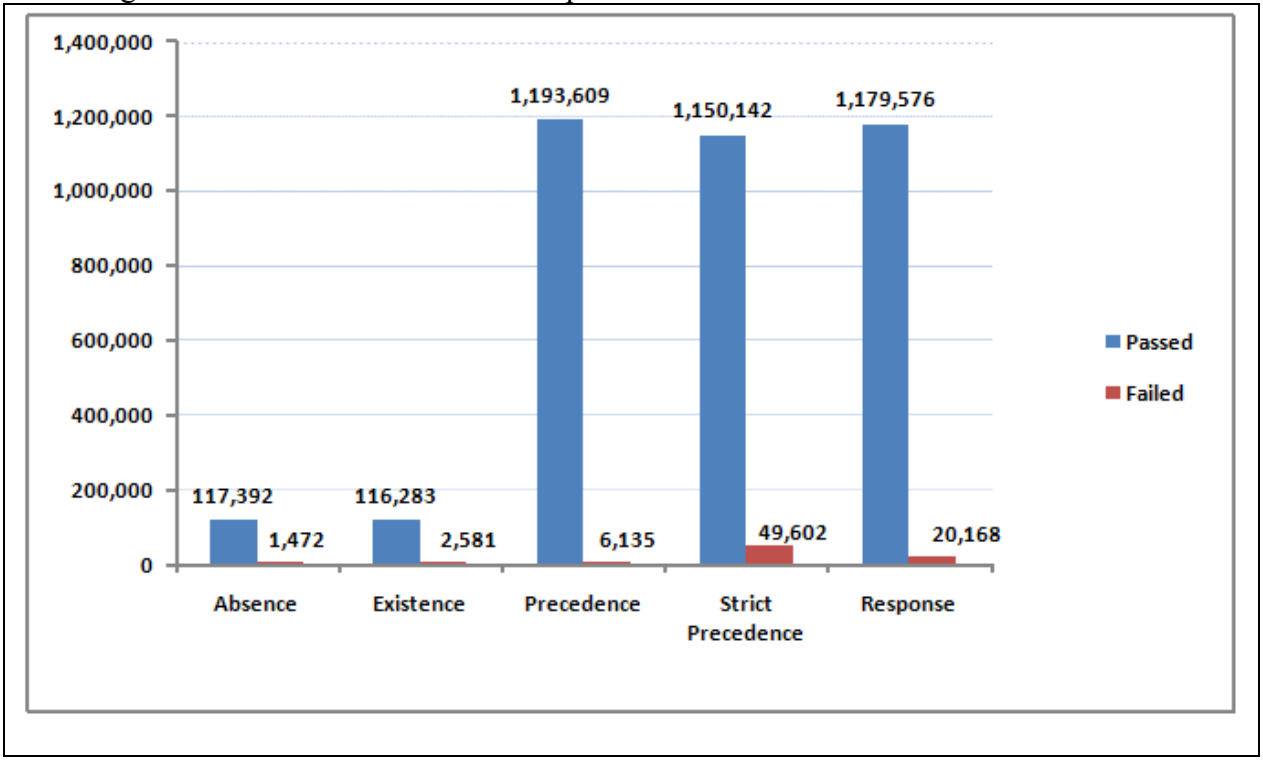


Figure 34: Number of Failing and Passing Test Cases.

The number of combinations in the *Precedence*, *Strict Precedence*, and *Response* patterns is greater and thus the number of test cases is also greater than the test cases in *Absence* and *Existence*. Notice that the *Strict Precedence* pattern has more failing test cases than *Precedence* and *Response* patterns. Figure 35 shows the percentage ratio between the passing and failing test cases.

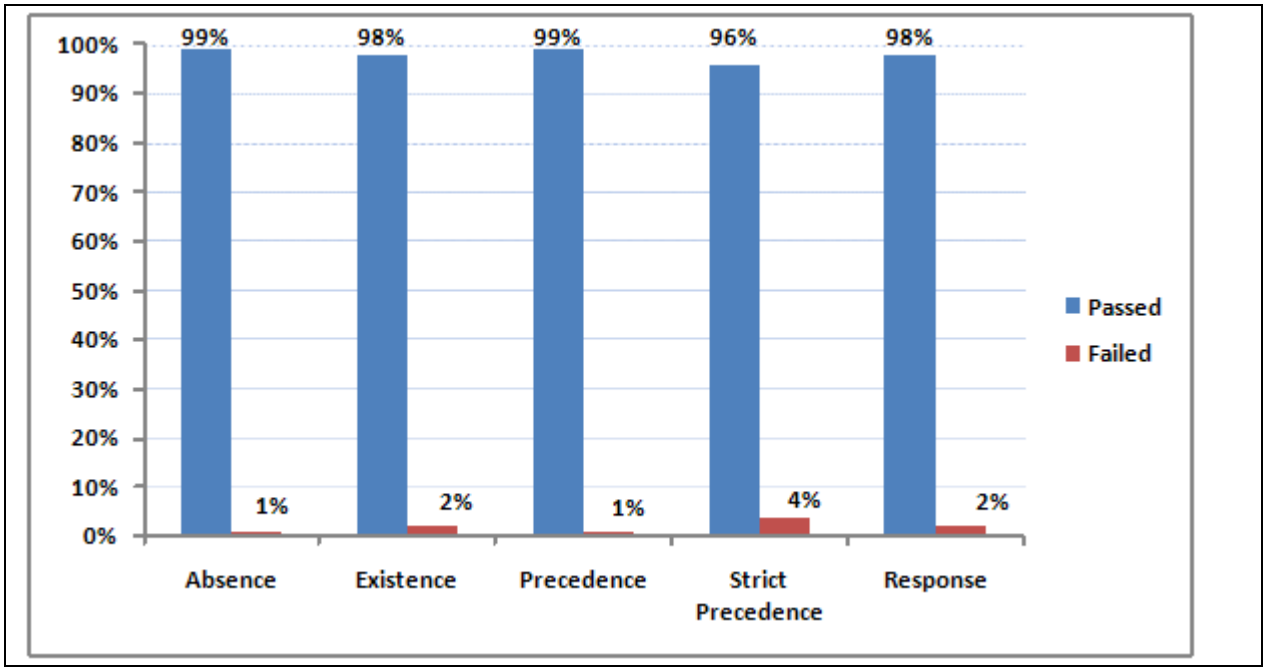


Figure 35: Ratio between Failing and Passing Test Cases.

## 5.2 Future Work

Although several significant test cases were generated, some of them were redundant. Redundant test cases are test cases that identify the same set of faults found by other test cases. Figure 36 shows a set of redundant test cases for the *Absence* pattern with *Before R* scope, and $AtLeastOne_C$ CP for both propositions.

**INPUT:**

**Pattern:** Absence
**Proposition P**: P1, P2, P3
**CP P**: $AtLeastOne_C$
**Scope**: Before R
**Proposition R**: R1, R2, R3
**CP R**: $AtLeastOne_C$

**TEST CASE SAMPLE:**

----------P1-P2-P3-R1-R2-R3---------- FALSIFIED

**REDUNDANT TEST CASES:**

1. ----------(P1)P2P3R1R2R3--------- FALSIFIED
2. ----------(P1P2)P3R1R2R3--------- FALSIFIED

Figure 36: Example of redundant test cases for the Before R scope.

The set of redundant test cases in Figure 36 clearly belongs to the test case sample. As long as one proposition *P* is before a proposition *R* in the execution trace, the result is satisfied. This is because of the condition that the CP for *P* is $AtLeastOne_C$. Therefore, it is redundant to have *P2* and *P3* propositions before *R1* if *P1* already exists before *R1*. Also, if we recall that the *Before R* scope can only be present in the execution trace once, there is no need to check for more R propositions (*R2* and *R3* as in 1 and 2 in the redundant test cases). The same idea can be applied to the *After L* scope.

The set of redundant test cases for the Global scope is similar to those of *Before R* and *After L*. The only difference is that as long as the proposition *P* exists (or does not exist for the *Absence* pattern), the execution trace is satisfied. Figure 37 shows an example of some redundant test cases in the *Global* scope.

**INPUT:**

**Pattern:** Absence
**Proposition P**: P1, P2, P3
**CP P**: $AtLeastOne_C$
**Scope**: Global

**TEST CASE SAMPLE:**

__ __ P1 __ __ __ FALSIFIED

**REDUNDANT TEST CASES:**

1. __ P1 P2 __ __ __ FALSIFIED
2. __ P2 __ __ __ FALSIFIED
3. __ P3 __ __ __ FALSIFIED
4. __ P1 P3 __ __ __ FALSIFIED

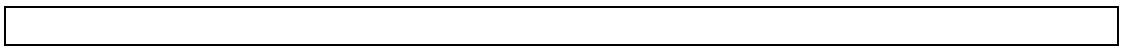

Figure 37: Example of redundant test cases for the Global scope.

Test cases 1 to 4 are redundant, because we are interested in at least one *P* in the execution trace. Therefore, once we find a violation in the execution trace, there is no need to keep checking for more propositions if we already found the violation (checking for *P2* when we already found *P1* in test case 1).

For the remaining scopes *After L until R* and *Between L and R*, the removal of redundant test cases is more complex. These two scopes can appear more than once in the execution trace. Once we find a violation in the trace, we stop looking for more propositions in the execution trace. The problem arises when the scope keeps holding in the execution trace. Therefore, we need to check the entire execution trace to verify that no violations appear. This is a problem that needs to be analyzed carefully to decide which test cases are redundant, because every execution trace seems to be significant for these two scopes.

# Appendix A: Main Interfaces in the LTL Verifier

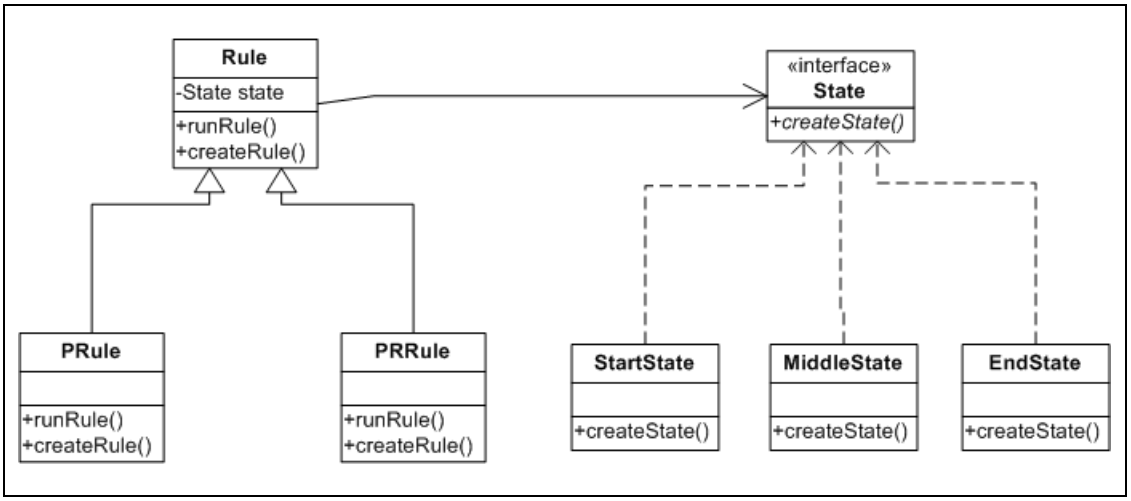


Figure 38: Rule and State generic interfaces.

In Figure 38, the Rule interface can be specialized into other objects besides *PRule* and *PRRule* objects. The same specialization can be done with the State interface. If the State object is not needed in the Rule object, the State object can be removed from the Rule interface or simply set to some null value.

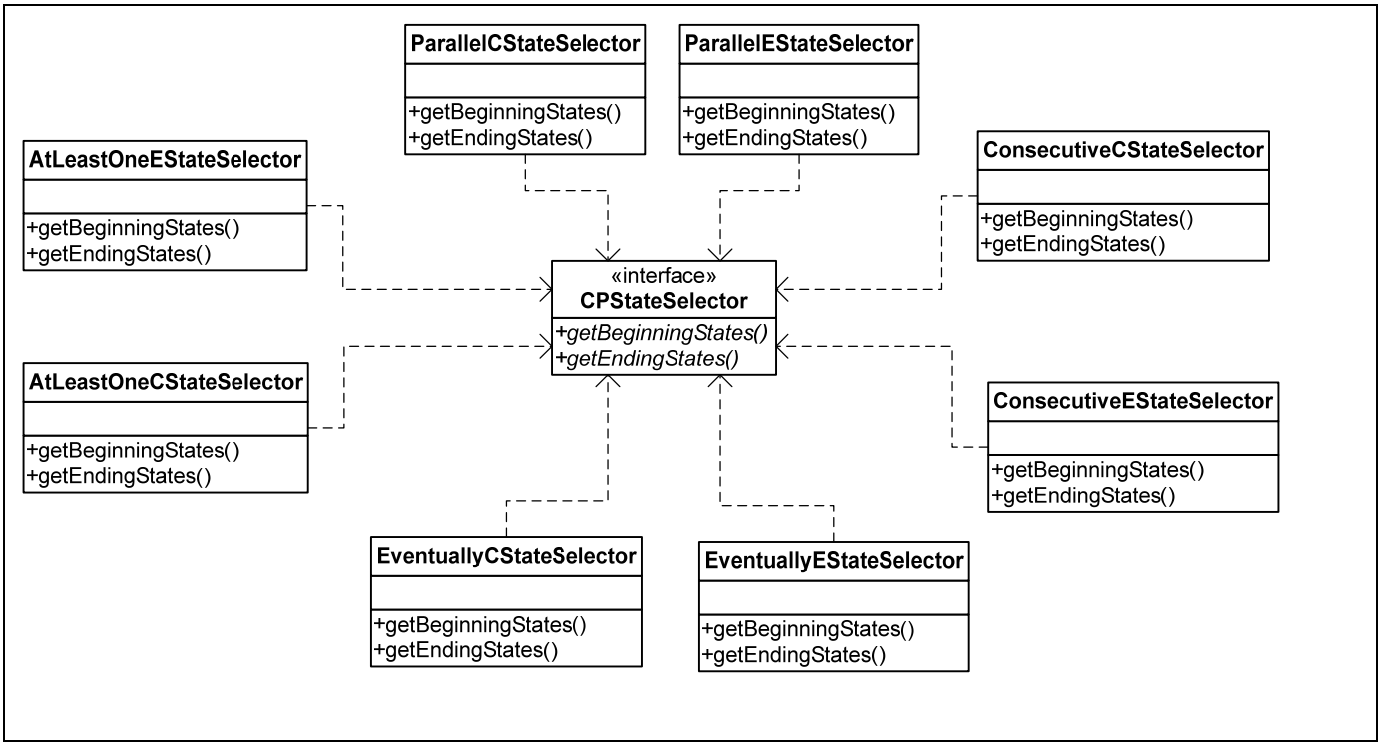


Figure 39: CPStateSelector.

The *CPStateSelector* (Figure 39) object belongs to the *ExecutionTraceParser* object. This interface provides a set of beginning and ending states for the CP classes to obtain their expected results (e.g. satisfied or falsified).

# References


AMMAN, P., AND OFFUIT, J. 2008. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK.

BALCI, O., NANCE, R. E., ARTHUR, J. D., AND ORMSBY, W. F. 2002. Improving the Model Development Process: Expanding Our Horizons in Verification, Validation, and Accreditation Research and Practice. In *Proceedings of the 34th Conference on Winter Simulation: Exploring New Frontiers,* San Diego, California, Dec 8 – 11. WSC '02. Society for Computer Simulation International, San Diego, CA, 653-663.

BARNAT, J., BRIM, L., JITKA, S. 2001. Distributed LTL Model-Checking in SPIN. In *Proceedings of the 8th International SPIN Workshop on Model Checking Software* (Toronto, Canada, May 21). SPIN '01. Lecture Notes in Computer Science (LNCS 2057), Springer-Verlag, New York, NY, 200-216.

BERNER, S., WEBER, R., AND KELLER, R. K. 2005. Observations and Lessons Learned from Automated Testing. In *Proceedings of the 27th International Conference on Software Engineering,* St. Louis, Missouri, May 15 – 21. ICSE '05. ACM Press, New York, NY, 571-579.

CHEON, Y. 2007. Automated Random Testing to Detect Specification-Code Inconsistencies. In *Proceedings of the 2007 International Conference on Software Engineering Theory and Practice,* Orlando, FL, USA, July 9-12, 2007. SETP-07. 112-119.

CIMATTI, A., CLARKE, E., GIUNCHIGLIA, E., GIUNCHIGLIA, M., PISTORE, M., ROVERI, S. R., AND TACCHELLA, A. 2002. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer-Aided Verification,* Copenhagen, Denmark, July 27-31, 2002. CAV '02, Springer Berlin/Heidelberg, New York, NY, 27-31.

DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. 1999. Patterns in Property Specification for Finite State Verification. In *Proceedings of the 21st International Conference on Software Engineering,* Los Angeles, California, May 16-22, 1999. ICSE '99, ACM Press, New York, NY, 411-420.

FRASER, G., AND WOTAWA, F. 2006. Property Relevant Software Testing with Model Checkers. In *Proceedings of the 2nd Workshop on Advances in Model-Based Software Testing,* Raleigh, North Carolina, November 7, 2006, 31(6). A-MOST '06. ACM Press, New York, NY, 1-10.

FRIEDMAN, N., HALPERN, J. Y., AND KOLLER, D. 2000. First-Order Conditional Logic for Default Reasoning Revisited. *ACM Transactions on Computational Logic* 1(2), 175-207.

GARCIA, L. A., 2007. Automatic Generation and Verification of Complex Pattern-Based Software Specifications. Master's Thesis. Department of Computer Science, University of Texas at El Paso. July, 2007.

GLENFORD, J. M. 2004. *The Art of Software Testing.* John Wiley and Sons, Inc, New Jersey, NJ.

GUNTER, E. L., 2003. From Natural Language to Linear Temporal Logic: Difficulties of Capturing Natural Language Specifications in Formal Languages for Automatic Analysis. In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation,* Chicago, Illinois, September 24-26, 2003. The Monterey Workshop Series.

HAMLET, D., AND MYERS, G. 1994. Foundations of Software Testing: Dependability Theory. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering,* New Orleans, Louisiana, USA, December 6-9, 1994. SIGSOFT '94. ACM Press, New York, NY, 128-139.

HIERONS, R. M. 2006. Avoiding Coincidental Correctness in Boundary Value Analysis. *ACM Transactions on Software Engineering and Methodology,* 15(3), 227-241.

HOLZMAN, G. J. 1997. The Model Checker SPIN. IEEE Transactions on Software Engineering, 23(5), 1-17.

HOLZMANN, G. J. 2003. *The SPIN Model Checker*. Addison-Wesley Professional, Boston, MA.

KANSOMKEAT, S., AND RIVEPIBOON, W. 2003. Automated-Generating Test Case Using UML Statechart Diagrams. In *Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology*, Pretoria, South Africa, September 17 – 19. SAICSIT '03. South African Institute for Computer Scientists and Information Technologists, Pretoria, South Africa, 296-300.

LEUCKER, M., AND SANCHEZ, C. 2008. Regular Linear Temporal Logic. In *Proceedings of the 4th International Colloquium on Theoretical Aspects of Computing*, Macao SAR, China, September 26 – 28, 2008. ICTAC '07. LNCS, Springer 2007, pp. 291-305.

LO, D., Siau-Cheng, K., AND CHAO, L. 2008. Mining Temporal Rules from Program Execution Traces. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, Washington, July 20 - 24, 2008. ISSTA '08. ACM Press, New York, NY, 50-56.

MILLER, A., DONALDSON, A., AND CALDER M. 2006. Symmetry in Temporal Logic Model Checking. *ACM Computing Surveys* 38(3).

MONDRAGON, O. 2004. Elucidation and Specification of Software Properties through Patterns and Composite Propositions to Support Formal Verification Techniques. Dissertation, Department of Computer Science, University of Texas at El Paso, May, 2004.

SALAMAH, S. I., GATES, A. Q., ROACH, S., AND MONDRAGON, O. 2005. Verifying Pattern-Generated LTL Formulas: A Case Study. In *Proceedings of the 12th International SPIN Workshop LNCS*, Springer, 2005, pp. 200-220.

SALAMAH, S. I. 2007. Generating Linear Temporal Logic Formulas for Complex Pattern-Based Specifications. Dissertation, Department of Computer Science, University of Texas at El Paso, May, 2007.

SHAHAMIRI, S. R., KADIR, W. M. N. W., AND MOHD-HASHIM, S. Z. 2009. A Comparative Study on Automated Software Test Oracle Methods. In *Proceedings of the 2009 4th International Conference on Software Engineering Advances*, Sofia, Bulgaria, July 26 – 29. ICSOFT '09. IEEE Computer Society, Washington, DC, 140-145.

SZTIPANOVITS, M., QIAN, K., AND Fu, X. 2008. The Automated Web Application Testing (AWAT) System. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, Auburn, Alabama, March 28 – 29, 2008. ACM-SE 46. ACM Press, New York, NY, 88-93.

VELA, C. Y. 2009. An Implementation for Automatic Generation of Linear Temporal Logic Formulas. Master's Project, Department of Computer Science, July 31, 2009.

# Vita

Cuauhtémoc Muñoz was born on February 10, 1985 in Ciudad Juarez, Chihuahua, Mexico. He obtained his High School Diploma from Preparatoria el Chamizal in 2002.During his senior year at High School, he was part of the Chamizal Solar Car Team that participated in the Winston Solar Car Challenge 2002 in Dallas, TX. In 2002, he was admitted at the University of Texas at El Paso. During his undergraduate studies, he worked as a volunteer at the Civil Engineering Department in conjunction with the Texas Department of Transportation. In his senior year, he was accepted by Dr. Roach as a Research Assistant in 2007. He obtained his bachelor's degree from the University of Texas at El Paso in December, 2007. In spring 2008, he was accepted to the Master's program in Computer Science at the University of Texas at El Paso. Cuauhtémoc worked as a Research Assistant during his first year as a graduate student. Thereafter, he formed part of the Teaching Assistant group and became the Lead Teaching Assistant in fall 2008. In the summer of 2009, he was an instructor for the class "Advanced Web Computing Techniques."

Permanent address: Canal de la Mancha 296

Ciudad Juárez, Chihuahua, MX, 32420

This thesis/dissertation was typed by Cuauhtémoc Muñoz.