CARL VON OSSIETZKY universität OLDENBURG

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

B.Sc. Computer Science

Bachelor Thesis

Natural Language Specifications for Safety-Critical Systems

written by

Benjamin Justice

Supervisors:

Prof. Dr. Andreas Winter
Dr. Tom Bienmüller

Oldenburg, November 20, 2013

Abstract

ISO 26262 prescribes the usage of formal methods for the development of safety-critical systems. A very effective concept is the formalization of requirement specifications written in natural language. The BTC EmbeddedSpecifier assists in this formalization process, whereby it requires intervention by the user. This thesis introduces a grammar and meta-model based approach for the formalization of natural language specifications. The approach, which is tailored towards the concept of the BTC EmbeddedSpecifier, translates the natural language specification into a semi-formal intermediate state. While the approach limits the spectrum of processable natural language specifications with its grammar, it is easily adaptable and extendable.

Contents

1 Introduction

This chapter briefly describes the content as well as the intent of this bachelors thesis. Section 1.1 gives an overview of the problem, which this thesis aims to solve. Section 1.2 describes the bachelor thesis goal and section 1.3 describes how the goal is achieved.

1.1 Problem Description

The development of safety critical systems such as cars is obliged to follow harsh industry standards. For the automotive industry, the industry standard ISO 26262 defines strict quality criteria for requirements specifications, test coverage, and other aspects of the software development process [Nat12]. To meet these requirements, the ISO 26262 defines a V-Model as a reference software development process [ISO12]. *Reactive Systems, Inc.* presented a simplified version of the V-Model in their whitepaper "Achieving ISO 26262 Compliance with Reactis", which is depicted in Figure 1.1 [Rea12].

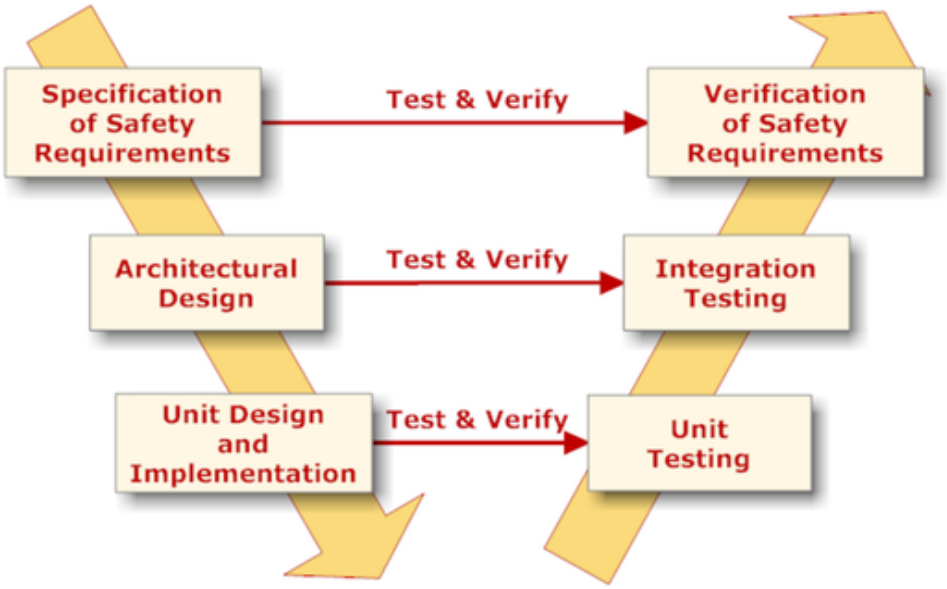

Figure 1.1: Simplified V-Model of the ISO 26262 standard [Rea12].

With the BTC EmbeddedSpecifier, BTC Embedded Systems offers software, which aids in developing safety critical systems while satisfying these requirements on the software development process. The BTC EmbeddedSpecifier aims to enable a broader user base to formulate requirements in a quality, which fulfills the needs of the ISO 26262. By abstracting the formal requirements towards

natural language, also qualified users, which are less knowledgeable of formal methods, are enabled to produce formal requirements. [BTC13].

The starting point for development within the development of safety-critical systems is often a *natural language specification*. In order to achieve a *semi-formal notation*, the user must structure his *natural language specification*, so that the requirement can be formalized more easily. *Semi-formal notation* within the BTC Embedded Specifier consists of patterns with solid semantics, which can be represented as automatons. To achieve this *semi-formal notation*, the user must chose an appropriate pattern from the BTC Pattern Catalog and map parts of the natural language requirement to predicates within the pattern. Further formalization requires the *semi-formal notation* to be connected to an architecture, such as Matlab Simulink.

In its current state, the BTC EmbeddedSpecifier requires several manual steps in order to achieve formal requirements. Section 3 offers an overview of this process. Prior to selecting a pattern and mapping predicates, the user must understand the formal semantics of the patterns in order to identify which patterns are compatible.

1.2 Problem Statement

Only very few employees in a company have the required knowledge to use formal methods by hand. By abstracting formal methods to a natural language level, these methods can be used by a broader range of employees. The gap between *natural language specification* and *semi-formal notation* is greater than the gap between *semi-formal notation* and *formal notation*. Thus, an opportunity to abstract the formalization process arises within the Derivation of a *semi-formal notation* from a *natural language specification*. In order to do so, one can attempt to automate two steps of the process: Selecting a compatible pattern as well as identifying variables to be mapped to the pattern.

1.3 Approach

This bachelor thesis aims to improve the automation of the process between *natural language specification* and *semi-formal notation*. To achieve this, a *grammar* will be defined, which can identify which patterns the *natural language specification* complies to. This grammar will be derived from fictional, aswell as real example requirements. Using a well-defined grammar, the *natural language specification*'s elements of interest could automatically mapped to predicates within the target pattern. This bachelor thesis will offer a prototype which can check if a natural language requirement is compliant to the grammar of BTC EmbeddedSystems AG's patterns and identify which pattern it complies to, as well as mapping the sentences elements of interest to the patterns predicates. Additionally, example sentences based on the grammar could be displayed to the user, so he can see which structure his natural language requirement must have, in order to conform to a given pattern.

2 Thesis Workflow

This chapter gives an overview of the workflow used to achieve the goal of this beachelor thesis. The workflow is visualized as an UML Activity Diagram in Figure 2.1. The workflow is organized in four segments: *Analysis*, *Prerequisites*, *Implementation* and *Validation*.

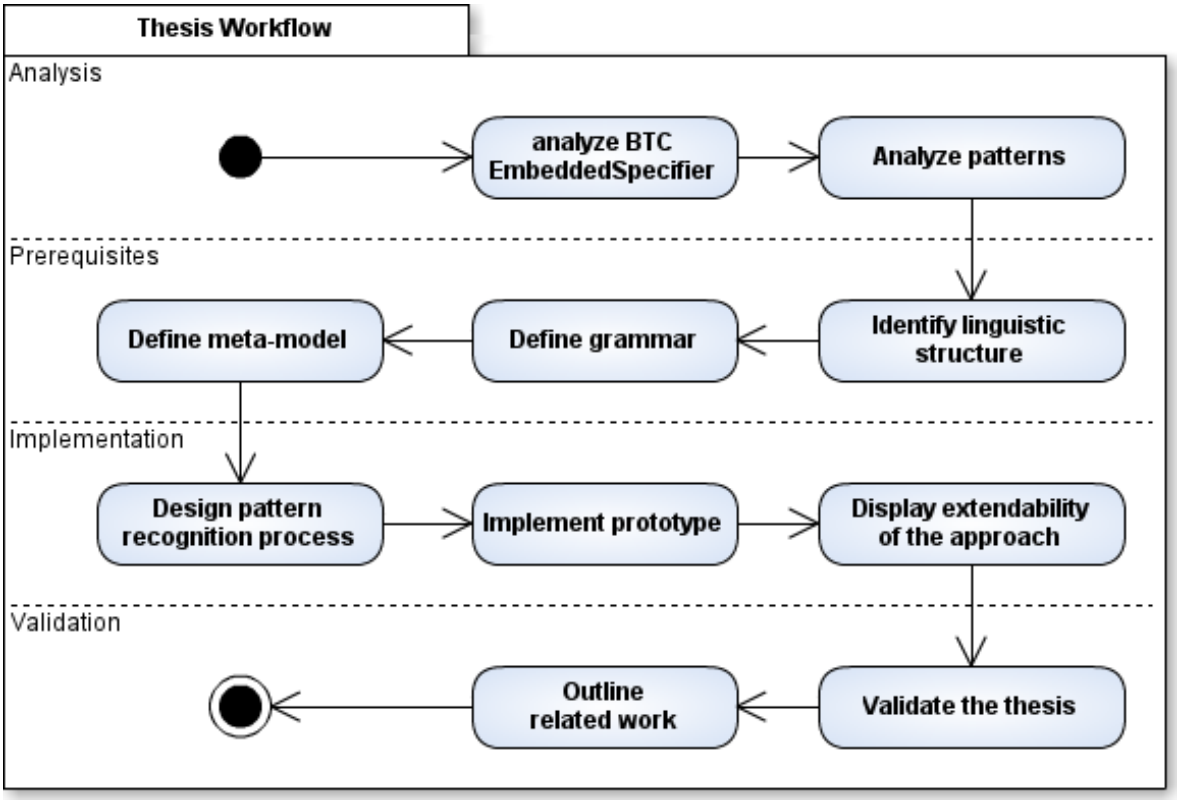

Figure 2.1: Thesis workflow organized in four segments.

The *Analysis* segment contains activities, which analyze the status quo. This includes an analysis of the *BTC EmbeddedSpecifier* in chapter 3 aswell as the *patterns* in chapter 4, which are used to represent semi-formal and formal requirements within the BTC EmbeddedSpecifier.

The *Prerequisites* segment adresses necessary prerequisites for the implementation of a pattern recognition process. This includes the *identification of linguistic structures* within the patterns, as well as the definition of a *grammar* and a *meta-model*. As the *identification of the linguistic structure* is bound to the patterns, it is included in chapter 4. Based on these linguistic structures, a *grammar is defined* in chapter 5. As a pattern recognition embedded in the grammar parsing process is not very extendable, a meta-model is defined in chapter 6. The meta-model allows for a cleaner pattern recognition implementation as well as a lucid extendability process.

The *Implementation* segment focusses on the implementation of the prototype for the thesis. Chapter 7 includes the *design of the pattern recognition process* based on the *grammar* and *meta-model*, as well as the *implementation of this process within the prototype*. The *extendability of the approach*, which it inherits from the grammar and meta-model, is displayed in chapter 8 along with an example.

The *Validation* segment concludes the thesis with a *validation* in chapter 9. The validation explains what was achieved within the thesis and what can be achieved in further research. Furthermore an *outline of related work* is given in chapter 9 as well.

3 The BTC EmbeddedSpecifier

This chapter describes the software BTC EmbeddedSpecifier by BTC Embedded Systems AG. Section 3.1 provides an overview of what the BTC Embeddedspecifiers does and how it affects software development. Section 3.2 offers an abstract explanation of the concepts behind the BTC EmbeddedSpecifier. Section 3.3 describes some example use-cases of the BTC EmbeddedSpecifier. Section 3.5 presents a simple workflow example to visualize the concepts and the usage of the BTC EmbeddedSpecifier. Section 3.6 offers a summary of the implemented and missing features of the BTC EmbeddedSpecifier. Lastly, section 3.7 gives a short overview of ISO 26262.

3.1 Overview

The BTC EmbeddedSpecifier is a software developed by BTC Embedded Systems to support the development of safety-critical systems which must fulfill industry standards. This chapter will regard ISO 26262, which is described in section 3.7, as an example industry standard. The ISO 26262 specifies itelf within a V-Model software development process [FER13]. The BTC EmbeddedSpecifier accompanies the software development in every phase of the V-Model as shown in Figure 3.1.

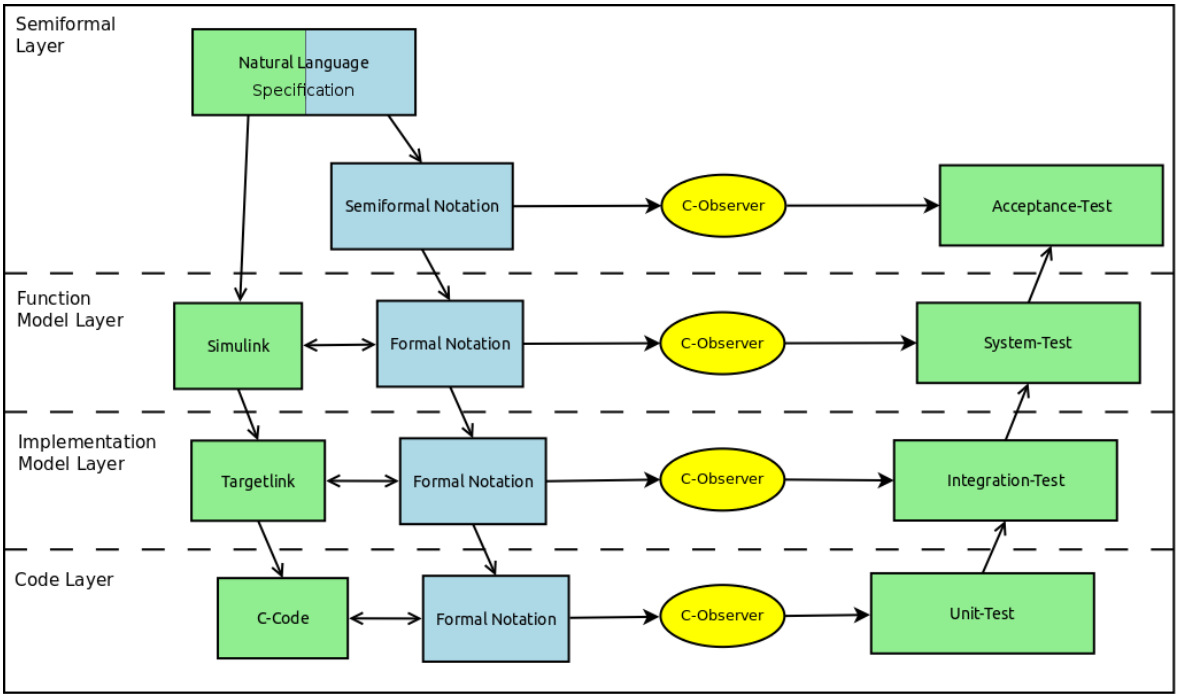

Figure 3.1: The V-Model in four layers with its artefacts (green), the BTC EmbeddedSpecifiers artefacts (blue) and its C-Observers(yellow)

The V-Model is split into four layers with dashed lines. Every layer contains artefacts which are represented by rectangles. The artefacts in green are created within development with the V-Model described in ISO 26262 [ISO12]. The BTC EmbeddedSpecifiers artefacts are depicted in blue and are parallel to the left-hand artefacts of the V-Model . In the *Semiformal Layer*, the V-Model's *Natural Languge Requirements* is extended by a *Semiformal Notation* on the BTC EmbeddedSpecifiers side.

In the *Functional Model* layer, aswell as in the *Implementation Model* and the *C-Code* layers, the BTC EmbeddedSpecifier generates a *Formal Notation* to the V-Model's corresponding artefacts. The BTC EmbeddedSpecifier generates *C-Observers*, which are represented by yellow ellipses. *C-Observers* are methods written in the C programming language which monitor the activity of a system in terms of violation of a requirement. The BTC EmbeddedSpecifiers workflow aims to generate C-Observers for each layer of the V-Model to offer a direct connection to the corresponding test artefacts on the right-hand side. In it's current version, this is only possible for the *Code Layer* .

3.2 Concept

The BTC EmbeddedSpecifiers workflow begins with requirements in form of a natural language specification and ends with the generation of C-Observers. This section will introduce the terminology [BTC], concepts and indivicual operations used by the BTC EmbeddedSpecifier by means of the abstract example shown in Figure 3.2 in five steps. A realistic workflow example is given in section 3.5.

In step one, the user chooses the requirements he wishes to formalize. In this case, the requirements is *"If it is dark, the lights turn on."*. The next step is to identify elements of interest within this requirement. These elements of interest are referred to as Macros. Macros are the variables within the natural language specification. For this requirement, the macros are *it is dark* and *lights turn on*. After the macros have been identified, a pattern must be created, in order to achieve a semiformal specification. This is done by mapping the requirement to a kernel pattern, which must be selected from a kernel pattern library. The BTC EmbeddedSpecifier requires an activation mode and a startup phase aswell, which are ignored in this example, but are explained in chapter 3.5. A kernel pattern is an automaton, which represents a logical sequence. In this case, the automaton must be able to represent *If x, then y*. The requirement is mapped to the kernel pattern via the macros. As can be seen in step two, the resulting pattern represents *If "it is dark", then "lights turn on"*. This pattern is not formal yet, because *it is dark* and *lights turn on* can be freely interpreted. They must be mapped to the interface of a concrete architecture, such as Matlab Simulink. Once the pattern is mapped to a concrete architecture, the pattern is unambiguous. The formal pattern represents *if (lightsensor == 0), then (light = 1)*. In order to generate a C-Observer, the BTC EmbeddedSpecifier requires a contract, which is created in step 4. A contract consists of a pattern which it refers to as commitment and an arbitrary number of patterns which it refers to as assumptions. If the systems state conforms to its assumptions, a contract guarantees, that the systems state also conforms to the commitment. The BTC EmbeddedSpecifier can generate a C-Observer for the contract in step four without further intermediate steps. In step five, the generated C-Observer is depicted. C-Observers can observe a systems state at runtime in order to report when the system violates the C-Observers contract, thus violating the underlying requirement. An overview of use cases for C-Observers is given in section 3.3

3.3 Common Use-Cases for the BTC EmbeddedSpecifier

A common *first step* with the BTC EmbeddedSpecifier for the customer is to realize that their requirements are not unambiguous enough to be formalized. The customer can then improve the quality of their requirements, eventually splitting requirements up into smaller requirements. By formalizing

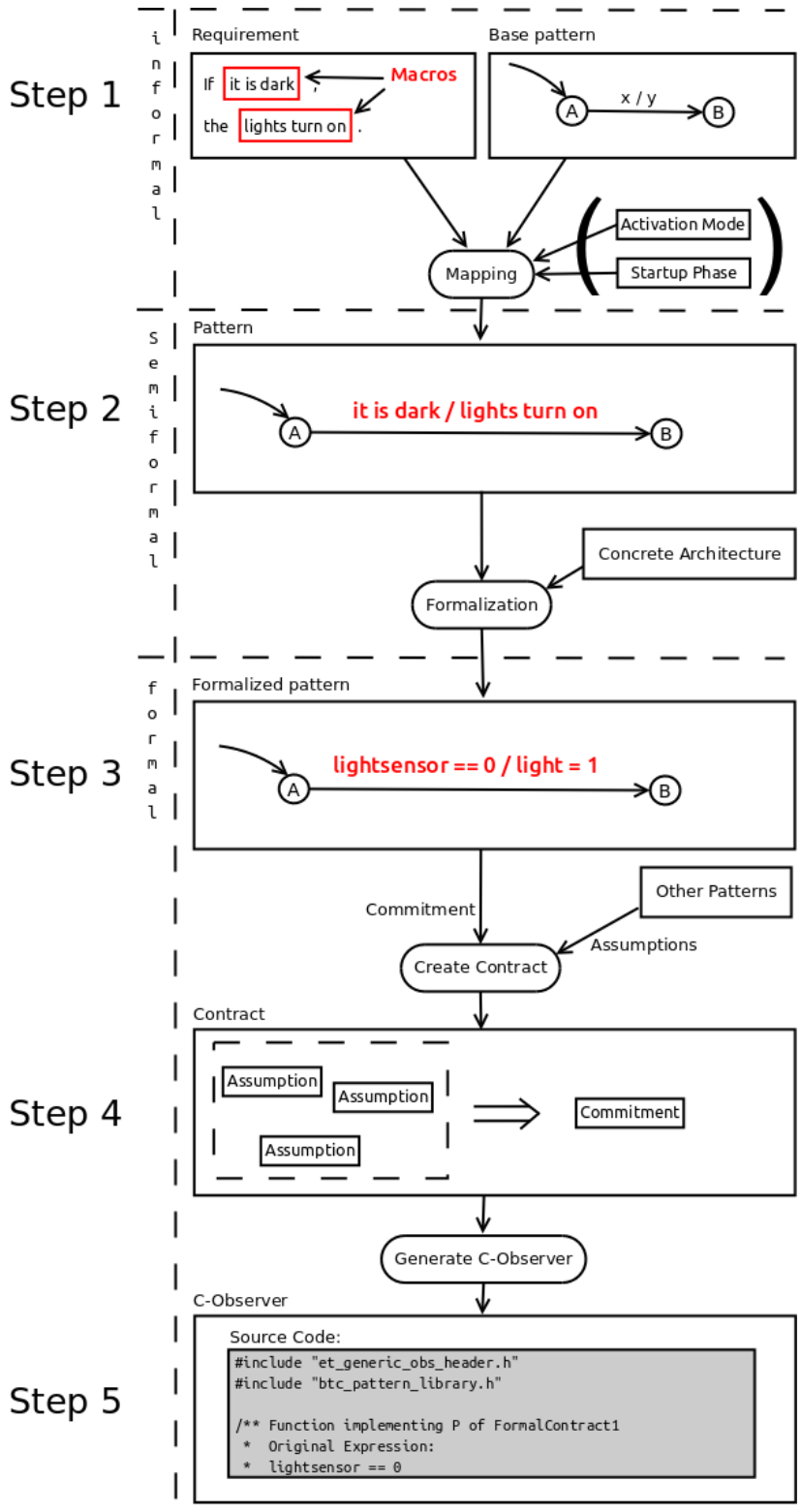

Figure 3.2: Abstact workflow from the natural language specification to the C-Observer

the requirements, the customer receives a formal notation for the functional and the implementation models aswell as for the C-Code to be used in the productive environment. This gives the customer opportunities for verification and tracibility measurements. The C-Observers can be used within a testing environment like the BTC EmbeddedTester to observe the test system and verify it. The C-Observers can also be used as components within the final product to observe the state of the system and give signals to other components if an exceptional state is observed. The formal notations and the C-Observers can be used in a versatile manner and are used differently by each customer.

3.4 The User Interface

The BTC EmbeddedSpecifier is based on the Eclipse Platform. After creating a new profile, the user can see the softwares user interface. The graphical user interface consists of four areas, as seen in Figure 3.3. The Profile Navigator in the *left area* allows for easy navigation within the profiles resources. After creating a new resource in the Profile Navigator or selecting an existing one, details about the resource are displayed in the *editor area* at the top. BTC EmbeddedSpecifier profiles have their requirements specification in the *Pool* folder. After importing an architecture, the architectures data is located next to the Pool. In addition to the details in the editor area, the user can see properties of the resource in the *bottom area*. The Interfaces of the resource can be seen on the right. If the resource is within a requirements specification, the Macros are displayed within the interface area. If the resource is within an imported architecture, macros are displayed along with signal interfaces defined by the architecture.

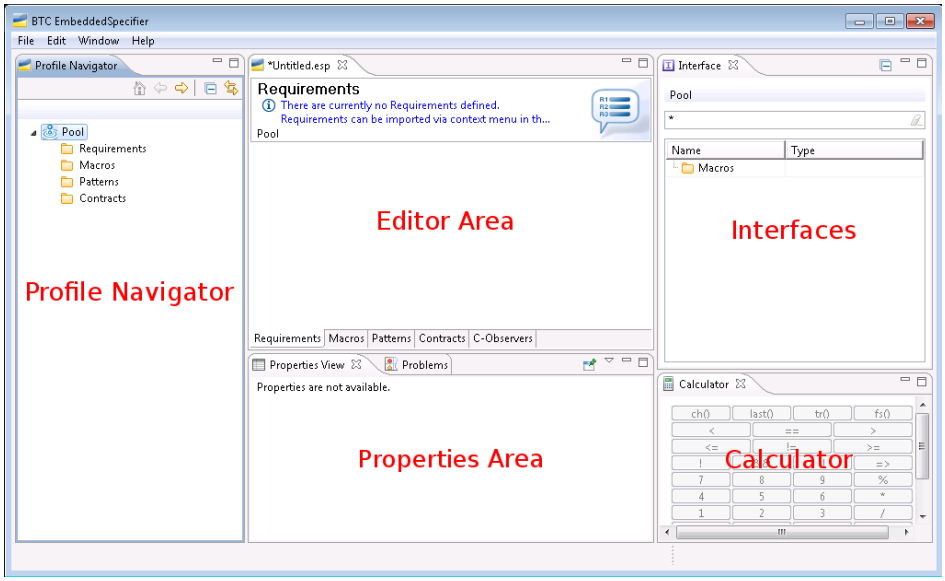

Figure 3.3: BTC EmbeddedSpecifier after creating a new profile

Source: BTC EmbeddedSpecifier

3.5 Workflow Example

This section will explain a simple workflow to obtain C-Observers from natural language requirements with the BTC EmbeddedSpecifier. The example used is provided together with the BTC EmbeddedSpecifier and is called *Power Window*. It contains natural language requirements for an electric window controller for cars. The example also contains a function model, an implementation model and the C-Code of the window controls, so that all four levels of the V-Model can be processed within this section. After creating a new project, the user must first import requirements, which is described in section 3.5.1. Section 3.5.2 will describe the process of bringing the natural language requirements into a semiformal form. In section 3.5.3 the concrete architecture will be imported and the semiformal requirement will become fully formalized. Finally Section 3.5.4 will cover the generation and use of C-Observers, which are the final artifacts of BTC EmbeddedSpecifier.

3.5.1 Importing Requirements

The first step while working with the BTC EmbeddedSpecifier is importing the requirements specification. The sources from which requirements specifications can be imported are IBM Doors as well as Microsoft Excel files. In this example the requirements specification is available as a Microsoft Excel file.

When importing a Microsoft Excel file, the user must configure how the BTC EmbeddedSpecifier should interpret the excel sheet's columns. After clicking on finish, the requirements are visible in the Profile Navigator in *Pool/Requirements/*. The user can click on a requirement to see the name and the description and verify that the requirements specification was imported correctly. This example will focus on requirement REQ_PW_4_1, which can be seen in Figure 3.4. The requirement's description is:
If the window moves up and an obstacle is detected, the window has to start moving down in less than 10ms.

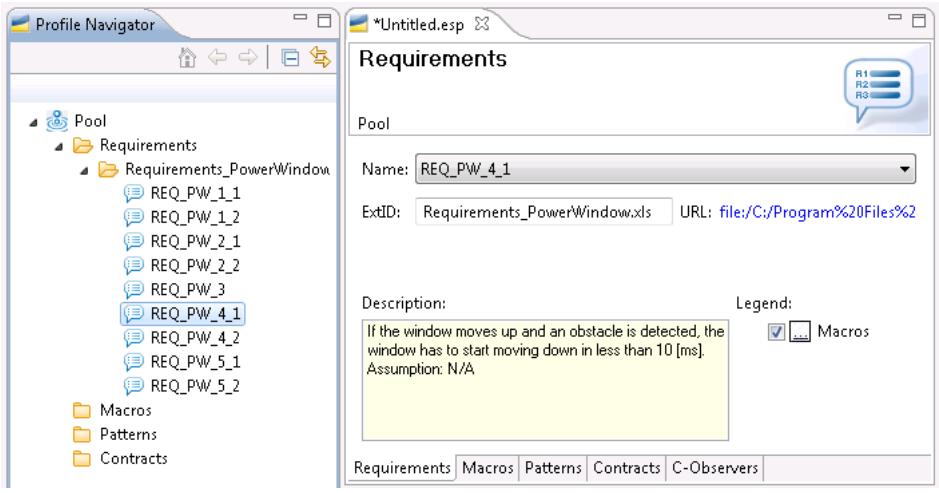

Figure 3.4: Imported Example Requirement in BTC EmbeddedSpecifier

Source: BTC EmbeddedSpecifier

3.5.2 Defining Semiformal Requirements

After the requirements were imported in section 3.5.1, they should be formalized. To achieve formal requirements, a semiformal intermediate stage is necessary. This section will explain how to create macros and patterns, which are required to achieve the semiformal stage.

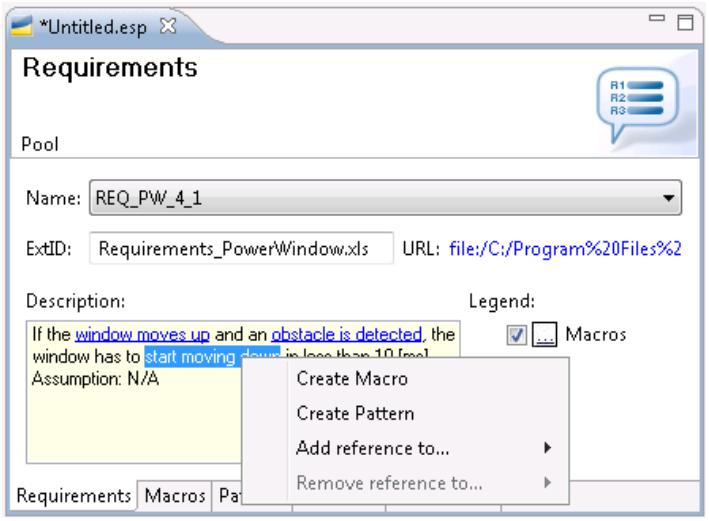

Figure 3.5: Creating macros from the natural language requirement

Source: BTC EmbeddedSpecifier

Creating Macros

Macros are the interface of the semiformal notation. They are linked to the formal notations interface in section 3.5.3. The macros are also used to map the natural language specification to the pattern in the next section. First the user choses the requirement he wishes to bring into a semiformal stage. In order to define macros the user must mark elements of interest. For REQ_PW_4_1 these elements of interest are *window moves up*, *obstacle is detected* and *start moving down*. The user must mark an element of interest within the requirements description, right click on it, and select “Create Macro” within the context menu as seen in Figure 3.5.

After doing so, the BTC EmbeddedSpecifier has linked the natural language specifications words to a macro. This step is to be repeated for every element of interest.

Creating a pattern

After all macros within the requirement have been identified, a pattern must be created for the requirement. Patterns are the formal representation for requirements within the BTC EmbeddedSpecifier [BTC12]. Patterns consist of a *kernel pattern*, a *start-up phase* and an *activation mode*. A *kernel pattern* is an automaton which contains the business logic of a pattern. BTC offers a broad selection of pre-defined kernel patterns for the user to chose from [BTC12]. The *start-up phase* contains in-

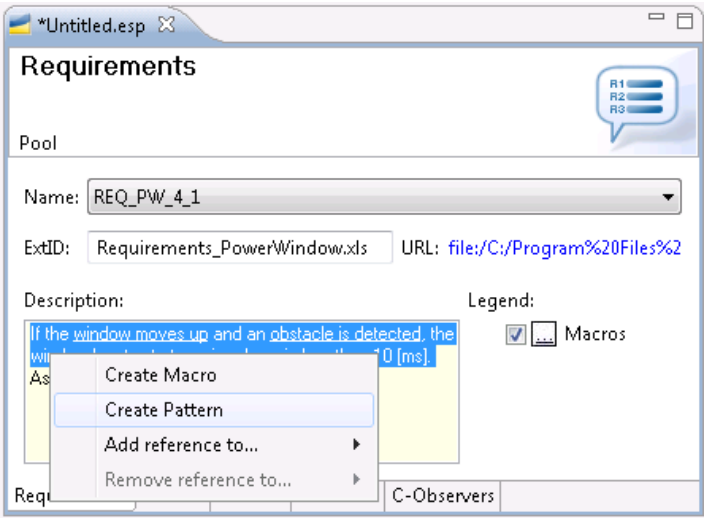

Figure 3.6: Creating a pattern from the natural language requirement

Source: BTC EmbeddedSpecifier

formation about when the pattern is activated. The *activation mode* determines, if the pattern can be activated repeatedly.

Specifying a pattern for the requirement is done by selecting a part of the description, right clicking, and selecting create pattern from the context menu as seen in Figure 3.6. In this example, the entire description must be selected. Note that *Assumption: [...]* is not selected. After creating the pattern, BTC EmbeddedSpecifier displays errors as depicted in Figure 3.7, because the new pattern is undefined. To define the pattern, the user must click on the **...-Button** next to the dropdown Menu after *Base Pattern*.

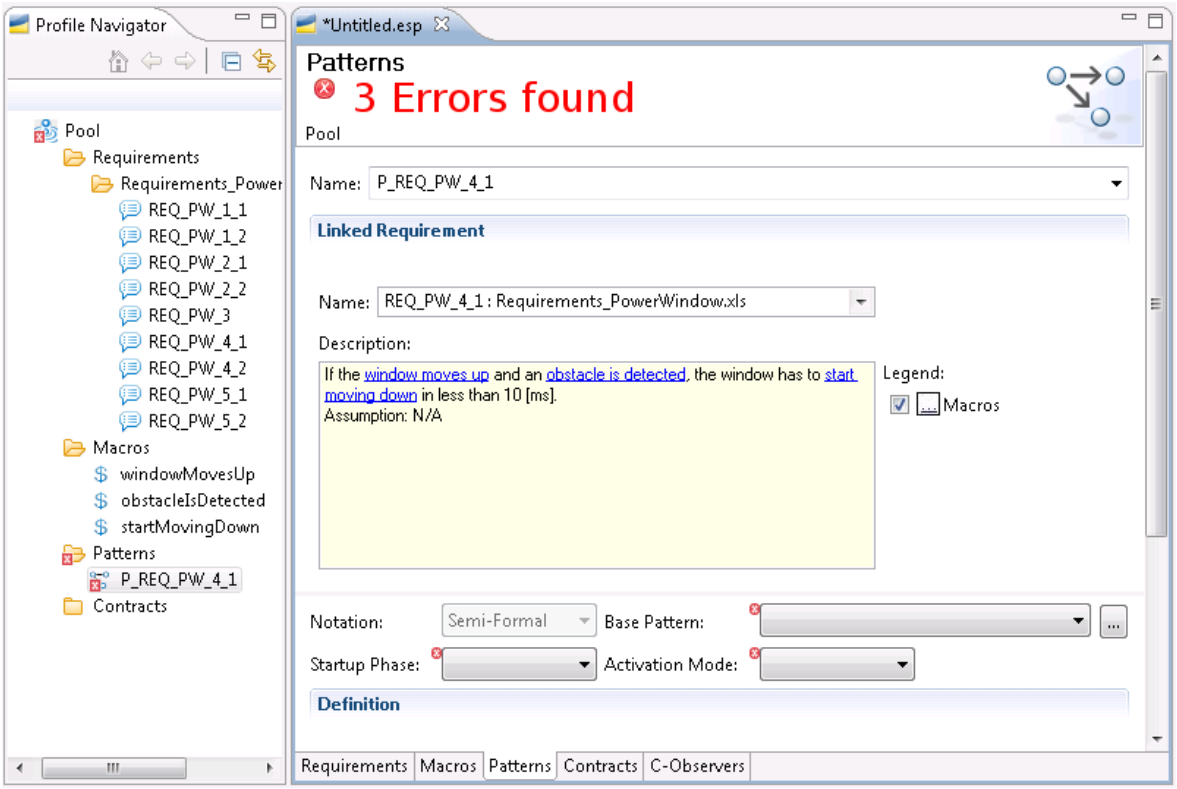

Figure 3.7: The newly created pattern in BTC EmbeddedSpecifier

Source: BTC EmbeddedSpecifier

This opens the BTC EmbeddedSpecifier's pattern library which can be seen in Figure 3.8. The kernel pattern library offers a selection of kernel patterns on the left. When a kernel pattern is chosen from the list, details about the kernel pattern are displayed including a timeline along with a description of the kernel pattern and the corresponding automaton. For each startup phase available in conjunction with the kernel pattern, a unique representation exists. Every kernel pattern contains variables, in which macros can be inserted into.

In this case the kernel pattern *P_implies_finally_Q_B* can represent the requirement REQ_PW_4_1, because REQ_PW_4_1, as an abstract statement, can be formulated as *If ("window moves up" and "obstacle detected") implies finally "start moving down" after "10ms"*. The user must additionally chose the startup phase and the activation mode. The startup phase *Immediately* matches this requirement, because the window control should be active after the system has started up. The activation mode for this requirement is *Cyclic*, because the requirement must be fulfilled repeatedly while the system is running. After selecting the activiation mode and start-up phase, the new kernel pattern, *cyclic_P_implies_finally_Q_B_after__N_steps*.

After a viable kernel pattern has been selected, the user must define boolean expressions for the variables defined in the kernel pattern. For this example the boolean expression for P is *$windowMovesUp && $obstacleIsDetected* and the boolean expression for Q is *$startMovingDown*. The variable *max_X* must be set to 0.010 seconds for this example requirement, as the statement must be true after a maximum time of 10 milliseconds. The BTC EmbeddedSpecifier now has a semiformal

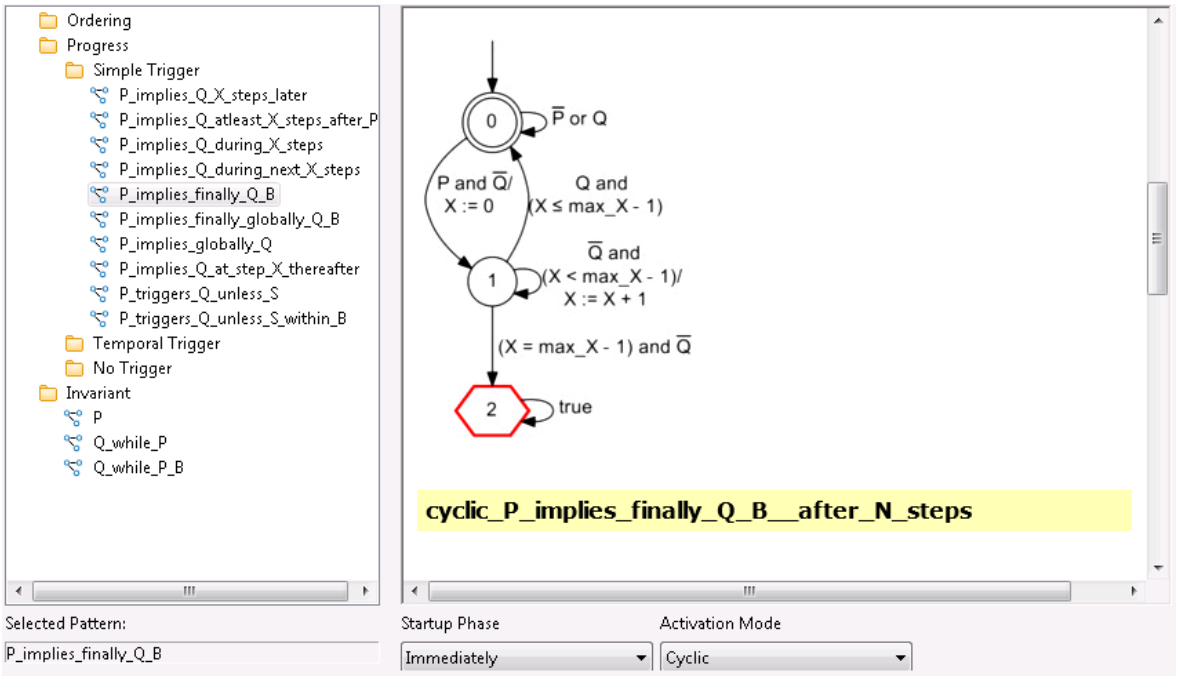

Figure 3.8: Selecting the kernel pattern from the kernel pattern library

Source: BTC EmbeddedSpecifier

representation of the requirement, because the automaton is a concrete model for the requirement. In order for a formal requirement to be present, the user must link the semiformal notation to a concrete architecture's interface, which is explained in section 3.5.3.

3.5.3 Formalizing the requirements

In order to achieve a formal representation of a requirement, the BTC EmbeddedSpecifier requires so-called contracts for the pattern's macros to be mapped to an architecture's signals. A contract consists of a single pattern as an assurance and an arbitrary amount of patterns as assumptions. If the systems state conforms to its assumptions, a contract guarantees, that the systems state also conforms to the commitment. Contracts are used internally by the BTC EmbeddedSpecifier to link the patterns to a concrete architecture. The user can import the EmbeddedTester profile from the PowerWindow Example to receive a concrete architecture, which the semiformal macros can be mapped to. After importing the architecture, the user must create a contract. For this example the user right clicks on the pattern *P_REQ_PW_4_1* in the profile navigator and selects create pattern.

The BTC EmbeddedSpecifier automatically creates a pattern without assumptions which has the pattern *P_REQ_PW_4_1* as its commitment as is visible in Figure 3.10. As this example requirement doesn't require any assumptions, the contract is complete. The contract along with its corresponding patterns and macros must be copied to the architecture. This is done by simply copying the contract and pasting it into the concrete architecture, which is the TargetLink Model in this case. As depicted in Figure 3.9, the BTC EmbeddedSpecifier will offer to copy the macros and patterns required by the contract aswell.

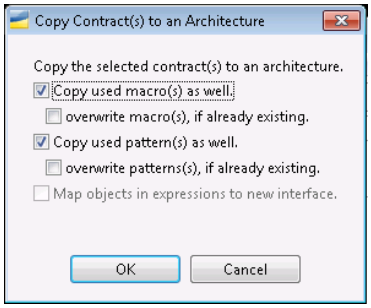

Figure 3.9: Copying contracts to an architecture

Source: BTC EmbeddedSpecifier

After confirming the copy operation, the user is confronted with errors within the architecture as seen in Figure 3.10. This is due to the interface of the architecture being undefined. The interface of the architecture, in contrast to the requirements specification, does not consist of Macros. The architecture has clearly defined input and output signals. The user must map the macros to the signals. This is done by selecting a macro within the architecture and entering a signal condition as seen in Figure 3.11. In this example the macros must be defined as in the following table:

Macro	**Definition**
windowMovesUp	move_up == 1
obstacleIsDetected	obstacle_detection == 1
startMovingDown	move_down == 1

The macros are automatically synchronized between the *TargetLink* and the *Production Code Host* folders within the BTC EmbeddedSpecifier. The BTC EmbeddedSpecifier now has a formal representation of the requirement. With this requirement, BTC EmbeddedSpecifier can generate a C-Observer as shown in section 3.5.4.

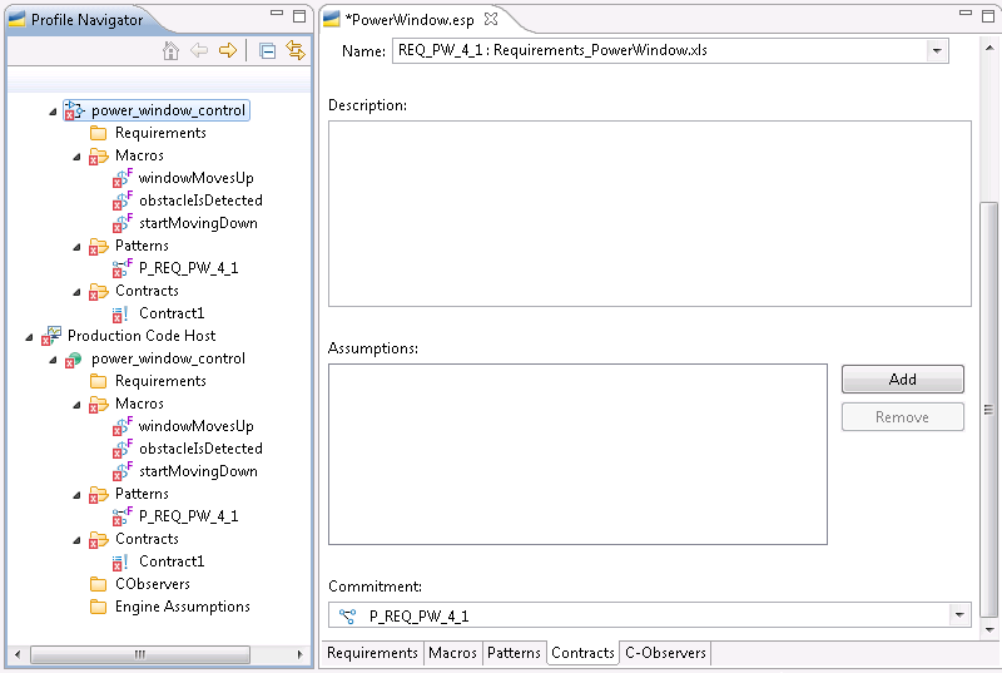

Figure 3.10: Erroneous requirements specification within the TargetLink architecture and C-Code

Source: BTC EmbeddedSpecifier

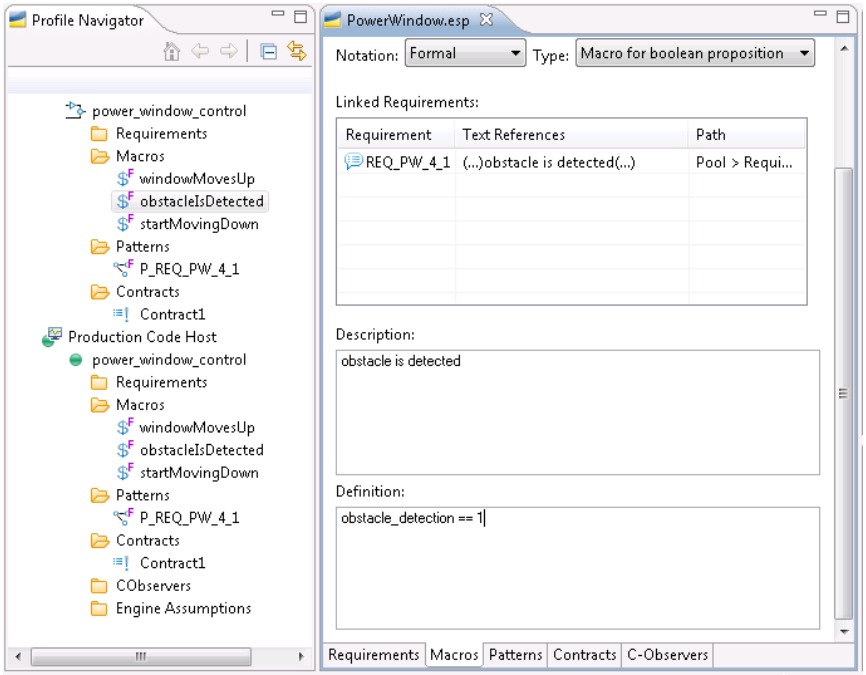

Figure 3.11: Formalizing the requirements specification via signal mapping

Source: BTC EmbeddedSpecifier

3.5.4 Generating a C-Observer

Having a formal requirement specification, the BTC EmbeddedSpecifier can generate a C-Observer for the contract which has *P_REQ_PW_4_1* as its commitment. C-Observers are Observers written in the C programming language and can observe the state of a system to recognize when its associated contract, i.e. the requirement which is the contracts commitment, has been violated. This contract is present within the *TargetLink* architecture and the *Production Code Host* architecture. In its current version, the BTC EmbeddedSpecifier can only generate C-Observers for the Production Code Host architecture. This is done by right-clicking the contract and selecting *Generate C-Observer* from the context menu. The BTC EmbeddedSpecifier will automatically generate the C-Observer. The C-Observers code can be viewed by selecting the C-Observer within the Profile Navigator.

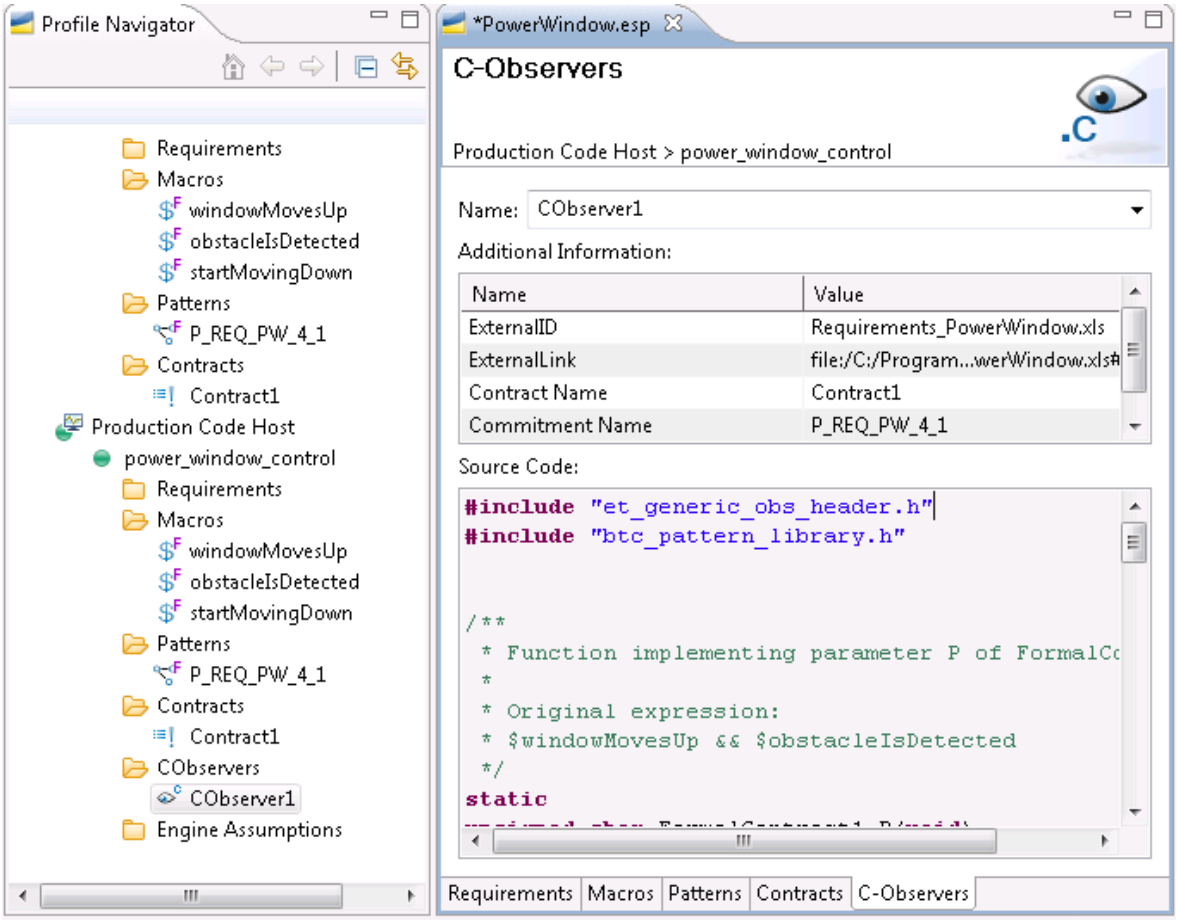

Figure 3.12: The generated C-Observer

Source: BTC EmbeddedSpecifier

3.6 Conclusion

As was described in section 3.5, the BTC EmbeddedSpecifier can assist the user in the process of translating a natural language specification to a formal requirement. These formal requirements can even be used to automatically generate tests for the target architecture. However the workflow from a natural language specification to a formal requirement requires many interventions by the user. This section also demonstrated, that the transition from a semi-formal to a formal requirement only requires a mapping to the target architecture. In contrast to this procedure, the transition from a natural language requirement to a semi-formal requirement is very ambiguous and thus requires experience with formal methods. A major improvement for the usability of the BTC EmbeddedSpecifier would be the abstraction or automation of the process from natural language specification to formal requirement. By doing so, the required experience with formal methods can be reduced by a significant amount.

3.7 ISO 26262

The ISO 26262 “Road Vehicles - Functional Safety” is an ISO norm which was released in 2011. It consists of ten parts and contains detailed regulations for the functional security of electronic systems within motor vehicles [ISO12]. While part one is an introduction and part ten is a glossary, parts two to eight describe the standard within the eight phases of a V-model. The norm adresses hazards which are caused by malfunctioning of the electronic system.

While mechanical damage is not adressed by ISO 26262, a human bodypart in between a closing window and similar hazards are adressed by the norm. Depending on several security aspects, the product is classified in ASIL levels (ASIL=automotive safety integrity level). There are four ASIL levels, from A (lowest) to D (highest). A product must fulfill at least ASIL A in order to become ISO 26262 certified [Nat12].

4 Patterns

Patterns were explained very roughly in section 3.5.2. BTC Embedded Systems AG uses patterns to represent semiformal and formal requirements in several products [BTC]. This chapter offers a more detailed view on patterns in section 4.1. Section 4.2 focusses on the kernel pattern, which is a patterns core component. It introduces the several kernel patterns in use by BTC EmbeddedSystems AG and defines natural language specifications for each kernel pattern. These natural language specifications are later used to validate the grammar-based approach proposed in section 1.3. Section 4.3 introduces the activation modes, whereas section 4.4 introduces the start-up phases.

4.1 Concept

Each pattern consists of an *activation mode*, a *Kernel Pattern* and a *start-up phase*. Every kernel pattern contains *Predicates*. These Predicates can be *Triggers*, *Actions* or quantities such as time intervals. *Triggers* are conditions, which must be fulfilled in order for *Actions* to take place. Figure 4.1 displays an example pattern, in which P is a *Trigger* and Q is an *Action*. *Initial* is the Activation Mode, *P_implies_Q_X_steps_later* is the Kernel Pattern and *immediate* is the Start-Up Phase. As is visible within this example, a patterns structure can be extracted from its name. The basic logic of the pattern is described in its kernel pattern. The activation mode and start-up phase influence the logic of the kernel pattern, resulting in a pattern which contains the kernel patterns logic, but can support cyclic occurrences of the kernel patterns logic, for example. Section 4.2 elucidates the concept of *Kernel Patterns* in detail and offers an overview of all available kernel patterns. Section 4.3 explains the term *Activation Mode* and introduces the possible activation modes. Section 4.4 explains the term *Start-Up Phase* and introduces the possible start-up phases.

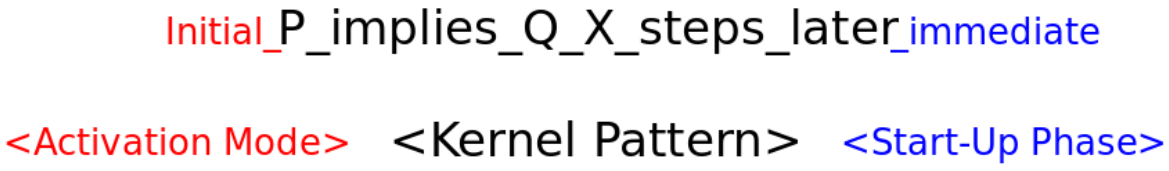

Figure 4.1: Composition of a patterns name.

4.2 Kernel Pattern

The *Kernel Pattern* is the main component of every pattern. It is represented by a Büchi automaton, which is an extension of automata, which can accept input of infinite length and contains an error state. Further details about Büchi automata are explained by Perrin [PP04].

The *Kernel Patterns* are classified by Triggers, which are depicted in yellow in Figure 4.2. The three main categories are *Progress*, *Invariant* and *Ordering* kernel patterns. Every category has several actions, which are marked in red. *Invariant* kernel patterns are described in section 4.2.1. Section 4.2.2 offers a detailed description of *Progress* kernel patterns, which are further divided into subcategories. Lastly section 4.2.3 introduces the *Ordering* kernel patterns. Predicates are depicted in bold font within the example requirements within this section. These examples were mostly created

by two english native speakers, who study computer science. Thus, most of these requirements are formulated based on some best practices from [PR11].

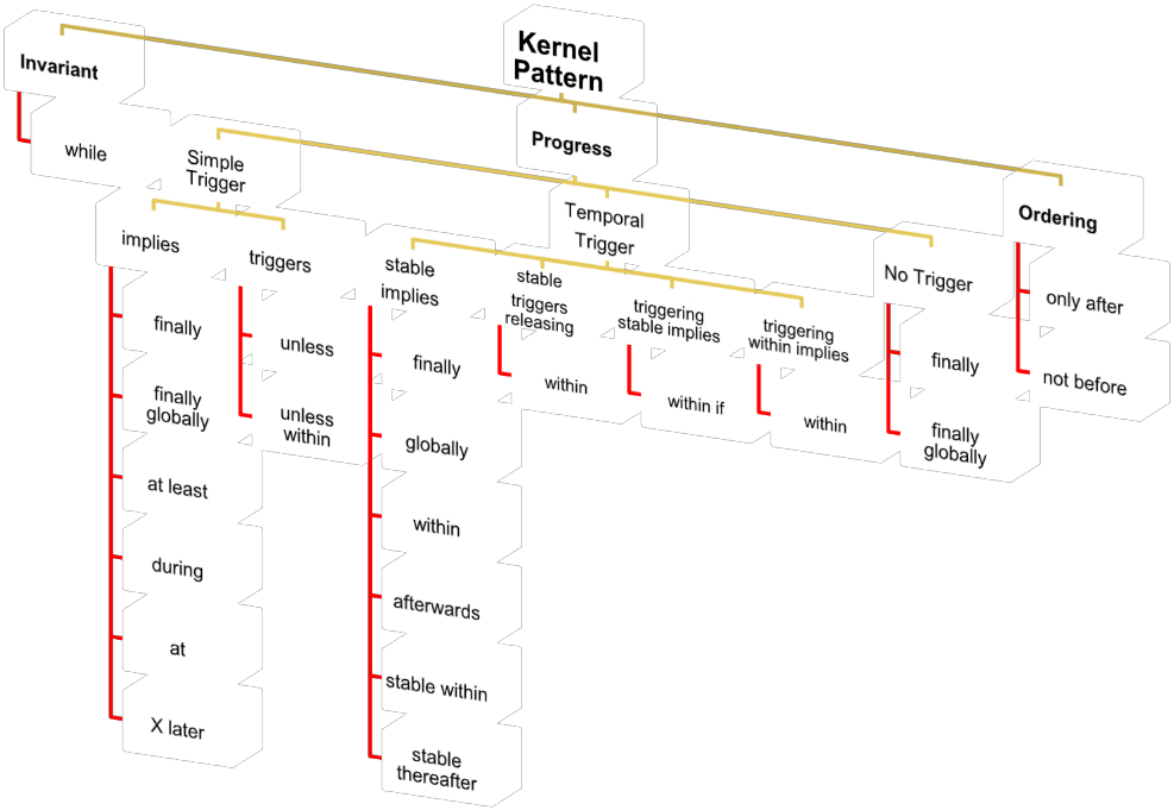

Figure 4.2: Classification of the kernel patterns. Triggers are represented in yellow, while actions are represented in red. [BTC]

4.2.1 Invariant

The *Invariant* kernel patterns provide an invariant condition which must be satisfied. This category contains a single kernel pattern type: *while*. This pattern type consists of two kernel patterns:

- **Q_while_P** specifies that if the condition P is true, then the condition Q must be true as well.

Kernel Pattern	**Q_while_P**
Examples	- **The wipers are active** while **it rains**. - **The parking sensors beep** while **an obstacle is detected**. - **The fuel display blinks** while **the fuel level is low**. - **The fuel display blinks** as long as **the fuel level is low**.
Common Structures	- **Q** while **P**. - **Q** as long as **P**.

- **Q_while_P_B** is analogous to *Q_while_P*, however it adds a time interval in which P must evaluate to false.

Kernel Pattern	**Q_while_P_B**
Examples	- **The wipers are active** for a maximum of **10 seconds** while **it rains**. - **The parking sensors beep** for a maximum of **10 seconds** while **an obstacle is detected**. - **The fuel display blinks** for a maximum of **10 seconds** as long as **the fuel level is low**.
Common Structures	- **Q** for a maximum of **X TIME** while **P**. - **Q** for a maximum of **X TIME** as long as **P**.

4.2.2 Progress

The *Progress* category is divided into three subcategories: *Simple Trigger*, *Temporal Trigger*, *No Trigger*. The following three subsections describe these subcategories and the corresponding kernel patterns.

Simple Trigger

The *Simple Trigger* category is subdivided into two categories: *triggers* and *implies*.

implies

The *implies* category contains eight kernel patterns, which cover the case of one expression implicating another expression.

- **P_implies_finally_Q_B** specifies that if the condition P is true, then the condition Q becomes true sometime within the next X steps.

Kernel Pattern	**P_implies_finally_Q_B**
Examples	- If **a crash is detected**, then **an emergency signal is sent** within **10 ms**. - **A detected crash** implies that **an emergency signal is sent** within **10 ms**.
Common Structures	- If **P**, then **Q** within **B TIME**. - **P** implies that **Q** within **B TIME**.

- **P_implies_finally_globally_Q_B** extends *P_implies_finally_Q_B* by the property, that Q must remain true forever.

Kernel Pattern	**P_implies_finally_globally_Q_B**
Examples	- If **a crash is detected**, then **an emergency signal is sent** continuously within **10 ms**. - **A detected crash** implies that **an emergency signal is sent** continuously within **10ms**.
Common Structures	- If **P**, then **Q** continuously within **B TIME**. - **P** implies that **Q** continuously within **B TIME**.

- **P_implies_globally_Q** specifies that if the conditions P and Q evaluate to true, then Q must remain true as long forever.

Kernel Pattern	**P_implies_globally_Q**
Examples	- If **a crash is detected**, then **an emergency signal is sent** continuously. - **A detected crash** implies that **an emergency signal is sent** continuously.
Common Structures	- If **P**, then **Q** continuously. - **P** implies that **Q** continuously.

- **P_implies_Q_atleast_X_steps_after_P** specifies that if the condition P is true, then the condition Q becomes true exactly X steps after P becomes false.

Kernel Pattern	**P_implies_Q_atleast_X_steps_after_P**
Examples	- If **the lights are on and it is dark**, then **the lights turn on** at least **1 second** after **the lights turn off and it is dark**. - **Lights on** and **darkness** imply that **the lights turn on** at least **1 second** after **the lights turn off and it is dark**.
Common Structures	- If **P** and **not Q**, then **Q** at least **X TIME** after **not P** and **not Q**. - **P** and **not Q** implies that **Q** at least **X TIME** after **not P** and **not Q**.

- **P_implies_Q_during_X_steps** specifies that if the conditions P and Q are true, then the condition Q remains true for the next X steps.

Kernel Pattern	**P_implies_Q_during_X_steps**
Examples	-If **it rains** and **the wipers are active**, then **the wipers are active** for **30 seconds**. -If **it rains** and **the wipers are active**, then **the wipers are active** during the next **30 seconds**. - **Rain** and **active wipers** imply that **the wipers are active** for **30 seconds**. - **Rain** and **active wipers** imply that **the wipers are active** during the next **30 seconds**.
Common Structures	- If **P** and **Q**, then **Q** for **X TIME**. - **P** and **Q** imply that **Q** for **X TIME**. - **P** and **Q** imply that **Q** during the next **X TIME**.

- **P_implies_Q_during_next_X_steps** specifies that if the condition P is true, then the condition Q becomes true for the next X steps.

Kernel Pattern	**P_implies_Q_during_next_X_steps**
Examples	-If **it rains**, then **the wipers are on** for **30 seconds**. -If **it rains**, then **the wipers are on** during the next **30 seconds**. -**Rain** implies that **the wipers are on** for **30 seconds**. -**Rain** implies that **the wipers are on** during the next **30 seconds**.
Common Structures	- If **P**, then **Q** for **X TIME**. - If **P**, then **Q** during the next **X TIME**. - **P** implies that **Q** for **X TIME**. - **P** implies that **Q** during the next **X TIME**.

- **P_implies_Q_at_step_X_thereafter** specifies that if the condition P is true, then the condition Q must be true exactly X steps later.

Kernel Pattern	**P_implies_Q_at_step_X_thereafter**
Examples	-If **it rains**, then **the wipers are on** exactly **30 seconds** later. -If **it rains**, then **the wipers are on** exactly **30 seconds** thereafter. -**Rain** implies that **the wipers are on** exactly **30 seconds** thereafter. -**Rain** implies that **the wipers are on** exactly **30 seconds** later.
Common Structures	- If **P**, then **Q** exactly **X TIME** later. - If **P**, then **Q** exactly **X TIME** thereafter. - **P** implies that **Q** exactly **X TIME** later. - **P** implies that **Q** exactly **X TIME** thereafter.

- **P_implies_Q_X_steps_later** specifies that if the condition P is true and the condition Q is not true, then the condition Q becomes true exactly X steps later.

Kernel Pattern	**P_implies_Q_X_steps_later**
Examples	- If **a crash is detected** and **the airbags are not released**, then **the airbags are released** exactly **10 ns** later. - **A detected crash** and **unreleased airbags** implies that **the airbags are released** exactly **10 ns** later.
Common Structures	- If **P** and **!Q**, then **Q** exactly **X TIME** later. - **P** and **!Q** imply that **Q** exactly **TIME** later.

triggers

The *triggers* category contains two kernel patterns, which cover the simultaneous fulfillment of two expressions, which imply an action until a third expression, which is mutually exclusive to one of the first two expressions, is satisfied.

- **P_triggers_Q_unless_S** determines that if the expression P is true, then either the expression Q or the expression S must be true in the same step. If Q was true during that step, then it must remain true until the expression S becomes true. Depending on whether P and Q or P and S are true, this pattern covers two distinct behaviors.

Kernel Pattern	**P_triggers_Q_unless_S**
Examples	- If **it is dark** and **the lights are on**, then **the lights are on** until **it is bright**. - If **the vehicle in front decelerates** and **the vehicle decelerates**, then **the vehicle decelerates** until **the safety distance is restored**.
Common Structures	- If **P** and **Q**, then **Q** until **S**.

- **P_triggers_Q_unless_S_within_B** is analogous to *P_triggers_Q_unless_S*, however it adds a time interval in which S must evaluate to true.

Kernel Pattern	**P_triggers_Q_unless_S_within_B**
Examples	- If **it is dark** and **the lights are on**, then **the lights are on** until **it is bright** within **10 minutes**. - If **the vehicle in front decelerates** and **the vehicle decelerates**, then **the vehicle decelerates** until **the safety distance is restored** within **2 seconds**.
Common Structures	- If **P** and **Q**, then **Q** until **S** within **B TIME**.

Temporal Trigger

The *Temporal Trigger* category is subdivided into four subcategories: *stable implies*, *stable triggers releasing*, *triggering stable implies*, *triggering within implies*.

stable implies

The *stable implies* category contains kernel patterns, which cover implications after signals have been stable for a given period of time.

- **P_stable_X_steps_implies_afterwards_Q** specifies that if the condition P is true for X steps, then Q must be true after exactly X steps. If P is not true for the entire duration of X steps, then the evaluation of Q is irrelevant.

Kernel Pattern	**P_stable_X_steps_implies_afterwards_Q**
Examples	- If **it rains** for **1 minute**, then **the wipers are activated**. - **Rain** for **1 minute** implies that **the wipers are activated**.
Common Structures	- If **P** for **X TIME**, then **Q**. - **P** for **X TIME** implies that **Q**.

- **P_stable_X_steps_implies_finally_Q_B** specifies that if the condition P is true for X steps, then Q must be true sometime within Y steps after the X steps are over. If P is not true for the entire duration of X steps, then the evaluation of Q is irrelevant.

Kernel Pattern	**P_stable_X_steps_implies_finally_Q_B**
Examples	- If **it rains** for **1 minute**, then **the wipers are activated** within **30 seconds**. - **Rain** for **1 minute** implies that **the wipers are activated** within **30 seconds**.
Common Structures	- If **P** for **X TIME**, then **Q** within **B TIME**. - **P** for **X TIME** implies that **Q** within **B TIME**.

- **P_stable_X_steps_implies_Q_within_Y_steps_unless_S** specifies that if the condition P is true for X steps, then Q must be true sometime within Y steps after the X steps are over. Q must remain true exactly until S is true. If P is not true for the entire duration of X steps, then the evaluation of Q is irrelevant.

Kernel Pattern	**P_stable_X_steps_implies_Q_within_Y_steps_unless_S**
Examples	- If **it rains** for **1 minute**, then **the wipers are activated** within **30 seconds** until **the windscreen is dry**. - **Rain** for **1 minute** implies that **the wipers are activated** within **30 seconds** until **the windscreen is dry**.
Common Structures	- If **P** for **X TIME**, then **Q** within **B TIME** until **S**. - **P** for **X TIME** implies that **Q** within **B TIME** until **S**.

- **P_stable_X_steps_implies_Q_stable_Y_steps_within_B_steps** specifies that if the condition P is true for X steps, then Q must be true for the duration of Y steps within B steps after the X steps are over.

Kernel Pattern	**P_stable_X_steps_implies_Q_stable_Y_steps_within_B_steps**
Examples	- If **it rains** for **1 minute**, then **the wipers are active** for **30 seconds** within **1 minute**.
Common Structures	- If **P** for **X TIME**, then **Q** for **Y TIME** within **B TIME**.

- **P_stable_X_steps_implies_globally_Q_within_Y_steps** specifies that if the condition P is true for X steps, then Q must start holding forever within Y steps after P.

Kernel Pattern	**P_stable_X_steps_implies_globally_Q_within_Y_steps**
Examples	- If **it rains** for **1 minute**, then **the wipers are active** continuously within **1 minute**.
Common Structures	- If **P** for **X TIME**, then **Q** continuously within **Y TIME**.

- **P_stable_X_steps_implies_Q_stable_Y_steps_B_steps_thereafter** specifies that if the condition P is true for X steps, then Q must start holding for Y steps within B steps after P.

Kernel Pattern	**P_stable_X_steps_implies_Q_stable_Y_steps_B_steps_thereafter**
Examples	- If **it rains** for **1 minute**, then **the wipers are active** for **30 seconds** exactly **1 minute** thereafter.
Common Structures	- If **P** for **X TIME**, then **Q** for **Y TIME** exactly **B TIME** thereafter.

stable triggers releasing

This category only contains the following kernel pattern:

- **P_stable_X_steps_triggers_S_releasing_Q_within_Y_steps** specifies that if the condition P is true for X steps, then Q must be true sometime within Y steps after the X steps are over unless S is true. Q must remain true until the expression S is observed as true simultaneously to Q.

Kernel Pattern	**P_stable_X_steps_triggers_S_releasing_Q_within_Y_steps**
Examples	- If **it rains** for **1 minute**, then **the wipers are active** within **30 seconds** until **the wipers are active** and **the windshield is dry**.
Common Structures	- If **P** for **X TIME**, then **Q** within **Y TIME** until **Q** and **S**.

triggering stable implies

This category only contains the following kernel pattern:

- **P_triggering_Q_stable_X_steps_implies_S_within_Y_steps_if_stable_T** specifies that after the condition P occurs, Q must be true for X steps. After X steps, T must become true until S becomes true. Within Y steps, T must become false or S must become true.

Kernel Pattern	**P_triggering_Q_stable_X_steps_implies_S_within_Y_steps_if_stable_T**
Examples	- If **the windshield is dirty**, then **the wipers are active** during the next **30 seconds**, after which **the wipers are active** for a maximum of **30 seconds** or until **the windshield is dry**.
Common Structures	- If **P** , then **Q** during the next **X TIME**, after which **S** within **Y TIME** or until **T**.

triggering within implies

This category only contains the following kernel pattern:

- **P_triggering_Q_within_X_steps_implies_S_within_Y_steps** specifies that after P occurs, if Q occurs within X steps, then S must occur within Y steps of Qs occurrence.

Kernel Pattern	**P_triggering_Q_within_X_steps_implies_S_within_Y_steps**
Examples	- If **a crash is detected**, then **the airbag is activated** within **5 ms**, after which **an emergency signal is sent** within **10 seconds**.
Common Structures	- If **P**, then **Q** within **X TIME**, after which **S** within **Y TIME**.

No Trigger

No Trigger kernel patterns contain actions, optionally with temporal constraints. They not dot require a trigger to be fulfilled, in order to perform their action.

- **P** merely specifies a predicate which is to evaluate to true.

Kernel Pattern	**P**
Examples	- **The electricity circuit is active**. - The light is on. - The display elements glow.
Common Structures	- **P**.

- **finally_P_B**

Kernel Pattern	**finally_P_B**
Examples	- **The motor is on** within **1 second**.
Common Structures	- **P** within **B TIME**.

- **finally_globally_P_B**

Kernel Pattern	**finally_globally_P_B**
Examples	- **The radio is powered** continuously within **5 seconds**.
Common Structures	- **P** continuously **B TIME**.

4.2.3 Ordering

The *Ordering* kernel patterns provide a solid order in which the conditions P and Q must occur. This category contains two kernel patterns:

- **Q_onlyafter_P** specifies that the condition Q may only be true, if the condition P has occurred before or parallel to Q.

Kernel Pattern	**Q_onlyafter_P**
Examples	- **The airbag is activated** only after **a crash is detected**.
Common Structures	- **Q** only after **P**.

- **Q_notbefore_P** specifies that the condition Q may only be true, if the condition P has occurred in a previous step.

Kernel Pattern	**Q_notbefore_P**
Examples	- **The airbag is not activated** before **a crash is detected**.
Common Structures	- **not Q** before **P**.

4.3 Activation Mode

The first component of a pattern is its activation mode. The activation mode defines the circumstances, under which the pattern must validate. The three available activation modes are *Initial*, *First* and *Cyclic*. Activation modes are relative to the system start-up phase, which is explained in section 4.4.

The *Initial* activation mode determines, that the pattern must be valid immediately after the start-up phase. The systems conformity to the pattern is checked solely 1 step after the start-up phase. The timeline for this activation mode is depicted in Figure 4.3.

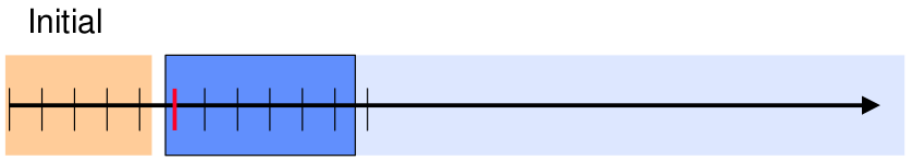

*Figure 4.3: Timeline of the **Initial** activation mode. [BTC12]*

The *First* activation mode determines, that the pattern must be valid once after the start-up phase. The systems conformity to the pattern is checked after the start-up phase until the patterns conditions are satisfied for the first time. The timeline for this activation mode is depicted in Figure 4.4.

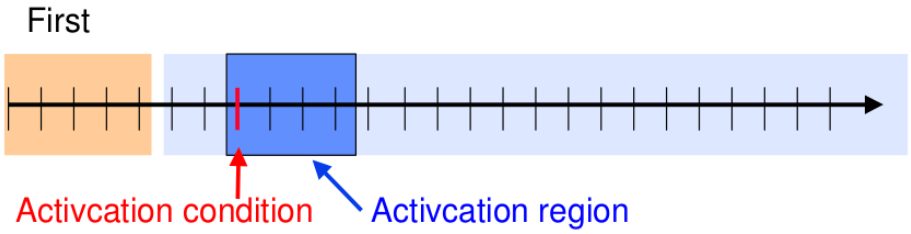

*Figure 4.4: Timeline of the **First** activation mode. [BTC12]*

The *Cyclic* activation mode determines, that the pattern must be valid repeatedly after the start-up phase. The systems conformity to the pattern is checked analogous to the *First* activation mode with the addition, that the conformity is checked again after the first occurrence. Hence, a *Cyclic* pattern's conditions are examined throughout the systems entire runtime after the start-up phase. The timeline for this activation mode is depicted in Figure 4.5.

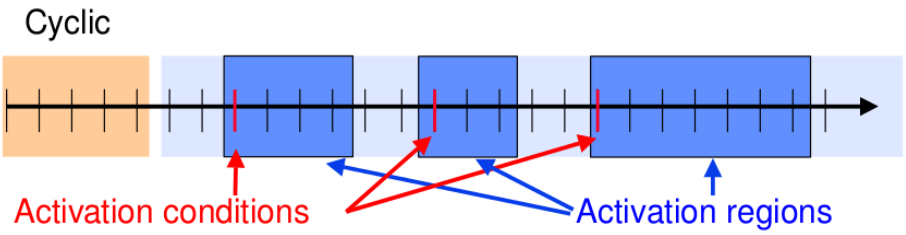

*Figure 4.5: Timeline of the **Cyclic** activation mode. [BTC12]*

4.4 Start-up Phase

The final component of a pattern is its start-up phase. The activation mode defines the precondition for a pattern to activate. The three available start-up phases are *Immediately*, *After N Steps* and *After Reading R*. A *Patterns* start-up phase precedes its *activation mode*.

The *Immediate* start-up phase determines, that the pattern has no start-up phase and is immediately active. Figure 4.6 depicts an empty timeline without a start-up phase.

*Figure 4.6: Timeline of the **Immediate** start-up phase. [BTC12]*

The *After N Steps* start-up phase determines, that the pattern is activated after a time interval has passed. Within the BTC EmbeddedSpecifier, the time interval can be specified as a time interval or a clock count. The timeline for this start-up phase is depicted in Figure 4.7

*Figure 4.7: Timeline of the **After N Steps** start-up phase. [BTC12]*

The *After Reading R* start-up phase determines, that the pattern is activated after an expression has evaluated to true. This expression can be used to determine when the engine of an automobile system has successfully started or to read other sensors within the system. The timeline for this start-up phase is depicted in Figure 4.8

*Figure 4.8: Timeline of the **After Reading R** start-up phase. [BTC12]*

5 Grammar

This chapter introduces the grammar which is used within the prototype in chapter 7. As proposed in section 1.3, a grammar is used in order to structure the input. As natural language is ambiguous, it is necessary to define a limited language to enable automatic mapping of macros. Section 5.1 explains the concept behind the grammar. Section 5.2 depicts the basic components, which constitute Kernel Patterns and defines common grammar rules. Section 5.3 outlines the common structures within the example requirements defined in section 4. Section 5.4 processes the outcome of the previous section to derive grammar rules. Section 5.5 examines and validates the constructed grammar.

5.1 Concept

The grammar within this chapter is to be used for the prototype in chapter 7. Thus, it must be able to parse a natural language specification based on this grammar. Furthermore the grammar rules may not overlap, so that the prototype can identify the applicable pattern by observing which grammar rules are used. By means of this grammar, it shall also be possible to identify macros, which can be mapped to a patterns predicates.

In order to create such a grammar, the kernel patterns are segmented by their triggers. Common Structures, which can be translated into grammar rules are identified for each trigger type. Activation modes and start-up Phases are implemented as prefix and suffix within the natural language specification.

5.1.1 Grammar Semantics

The grammar semantics used within this chapter are conform to the **Extended Backus-Naur Form (EBNF)** as described in ISO 14977:1996 [ISO96] with the addition, that Non-Terminals are written in uppercase. This definition for EBNFs includes the following basic rules:

Function	Symbol	Example
Terminal symbols	"Symbol"	"a" "house" "continuously"
Nonterminal symbols	SYMBOL	NUMBER DIGIT ACTION TRIGGER
Rule End	;	"a";
Concatenation	,	ACTION, TRIGGER
Definition split	|	ACTION | ACTION, TRIGGER
Optional items	[ ]	[TRIGGER], ACTION, [TIME]
Group items	( )	(Trigger, Action) | Action
Repetition (0 or more times)	{ }	{"a"}, "bb", "c"
Multiplication	*	{"a"}, 2*"b", "c"
Exclusion	-	NUMBER - "3" {"a"}- *This excludes the empty word*

5.2 Basic Structure of all Patterns

The grammar rules within this section are derived from parse trees of each kernel pattern. These parse trees, as well as the resulting grammar rules are restricted to the corresponding kernel pattern. As such, the grammar rules contain much redundancy when regarded together. The structures used for these parse trees are extracted from the example requirements in section 4.2.

All Kernel Patterns consist of a *Trigger*, which must be satisfied and is then followed by an *Action*. Many Kernel Patterns require additional Temporal Constraints. An example of such a Temporal Constraint exists within the example of *finally_P_B*: "after 1 second at the latest". Together with Trigger and Action, Time constitute the basic structure, which is shared by all Kernel Patterns and can be seen in Figure 5.1.

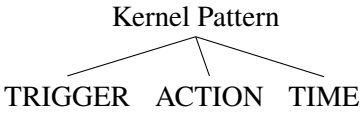

Figure 5.1: Abstract parse tree for Kernel Patterns.

This parse tree represents the following grammar rules:

```
KERNEL_PATTERN = TRIGGER, ACTION, TIME;
```

5.2.1 Action and Trigger

Most Triggers and Actions are based upon *Simple Actions* and *Simple Triggers*, which are grammatically equivalent and represent *Predicates*. For readability purposes, the grammar will include the *intermediate* rules *Simple Action* and *Simple Trigger*.

Figure 5.2 displays these two rules as parse trees.

Figure 5.2: Parse trees for ACTION and TRIGGER

These parse trees represent the following grammar rules:

```
ACTION = SIMPLE_ACTION;
SIMPLE_ACTION = PREDICATE;
TRIGGER = SIMPLE_TRIGGER;
SIMPLE_TRIGGER = PREDICATE;

PREDICATE = UNINTERPRETED STRING;
// A common string in the context of computer science.
```

5.2.2 Interval

Furthermore many Kernel Patterns require time intervals. The rule for a time interval is the same in all cases, as is depicted in figure 5.3

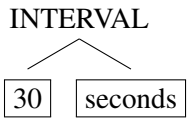

Figure 5.3: Parse tree for INTERVAL

These parse trees represent the following grammar rules:

```
INTERVAL = NUMBER, TIME_UNIT;
NUMBER = DIGIT | {DIGIT}, DIGIT;
TIME_UNIT = ("nanosecond" | "millisecond" | "second" | "minute" |
            "hour" | "day" | "step"), [s];
```

5.2.3 Activation Mode

An activation mode defines, under which conditions a pattern must validate. It determines, if the pattern must validate immediately after the patterns start-up phase, if the pattern must validate once any time after the start-up phase, or if the pattern must validate repeatedly any time after the start-up phase. The activation mode can be linguistically represented as a prefix.

5.2.3.1 Cyclic

As a natural linguistic sentence does not constrain its occurences, the *cyclic* activation mode does not contain a prefix.

5.2.3.2 Initially

Initially determines, that a kernel patterns logic applies directly after the start-up phase has ended. The grammar rules are depicted in Figure 5.4

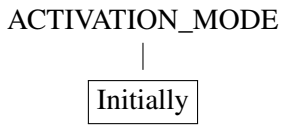

Figure 5.4: Parse tree for INITIAL

The resulting grammar for the prefix is as follows:

```
ACTIVATION_MODE = "Initially";
```

5.2.3.3 First

Initially determines, that a kernel patterns logic applies a single time, sometime after the start-up phase has ended. The grammar rules are depicted in Figure 5.5.

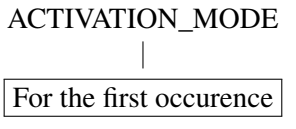

Figure 5.5: Parse tree for FIRST

The resulting grammar for the prefix is as follows:

```
ACTIVATION_MODE = "For the first occurence";
```

5.2.4 Start-Up Phase

The start-up Phase resembles a condition, which must be met, before the kernel patterns logic applies. As such, it can be applied as a suffix to the kernel patterns grammar. The linguistic representation for a start-up phase can be derived from the names of the start-up phases.

Immediate

As the *Immediate* start-up phase represents an empty start-up phase, a pattern implementing the *Immediate* start-up phase does not contain a suffix.

After N Steps

After N Steps determines, that a kernel patterns logic applies n steps after start-up. The grammar rules can be directly derived from the start-up phases name and are depicted in Figure 5.6.

The resulting grammar for the suffix is as follows:

```
START-UP = AFTER_WORD, INTERVAL, COMMA;
AFTER_WORD = "After";
COMMA = ",";
```

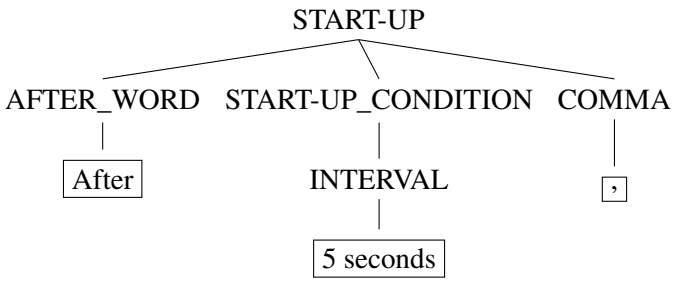

Figure 5.6: Parse tree for After N Steps

After Reaching R

After N Steps determines, that a kernel patterns logic applies after an expression has evaluated to true. This expression can be translated to a predicate and thus, be easily integrated with the predicate based grammar. The grammar rules are depicted in Figure 5.7

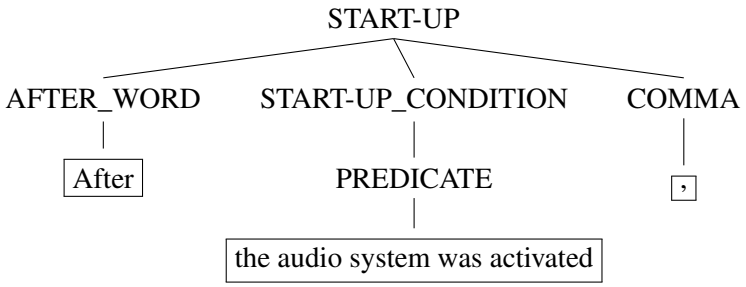

Figure 5.7: Parse tree for After N Steps

The resulting grammar for the suffix is as follows:

```
START-UP = AFTER_WORD, PREDICATE, COMMA;
AFTER_WORD = "After";
COMMA = ",";
```

5.3 Common Structures within Example Requirements

As the common structures displayed in section 4.2 are not detailed enough to create a grammar, the grammatical structure of the example sentences are used here to identify common structures in more detail. The kernel patterns are segmented by their triggers, as depicted in Figure 4.2 in section 4.2. The Kernel Patterns are grouped by their triggers and analyzed in the following order:

- Section 5.3.1: Invariant
- Section 5.3.2: Simple Trigger (implies)
- Section 5.3.3: Simple Trigger (triggers)

- Section 5.3.4: Temporal Trigger (stable implies)
- Section 5.3.5: Temporal Trigger (stable triggers releasing)
- Section 5.3.6: Temporal Trigger (triggering stable implies)
- Section 5.3.7: Temporal Trigger (triggering within implies)
- Section 5.3.8: No Trigger
- Section 5.3.9: Ordering

5.3.1 Invariant

Invariant Kernel Patterns consist of of *ACTION* and *TIME* components, whereas the *TIME* components contain a *SIMPLE_TRIGGER*.

Q_while_P

Q_while_P is comprised of an *ACTION* and a *TIME* constraint. This *TIME* consists of the *WHILE* rule, which is represented by "while TRIGGER". The resulting parse tree, which can identify "ACTION while TRIGGER", is depicted in Figure 5.8.

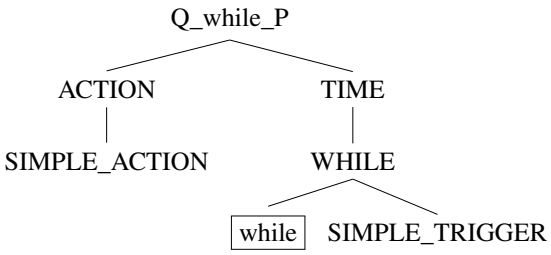

Figure 5.8: Parse tree for Q_while_P

This parse tree represents the following grammar rules:

```
ACTION = SIMPLE_ACTION;
TIME = WHILE;
WHILE = "while", SIMPLE_TRIGGER;
```

Q_while_P_B

Q_while_P_B is comprised of an *ACTION* and a *TIME* constraint. The *TIME* constraint equals the previous *Kernel Pattern*. The *ACTION* is extended by a *MAXIMUM_INTERVAL* rule, which defines a time limit for the action. The resulting parse tree, which can identify "ACTION for a maximum of INTERVAL while TRIGGER", is depicted in Figure 5.9.

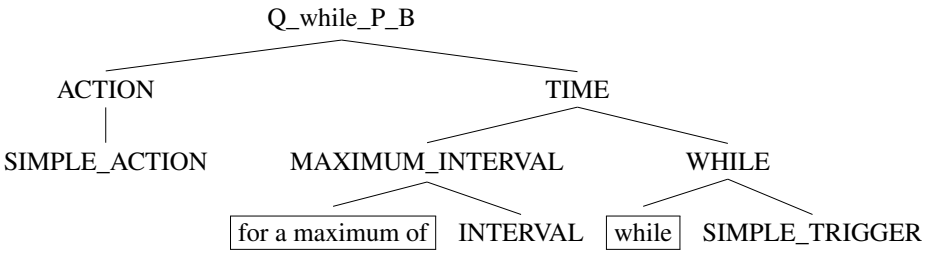

Figure 5.9: Parse tree for Q_while_P_B

This parse tree represents the following grammar rules:

```
ACTION = SIMPLE_ACTION, MAXIMUM_INTERVAL;
MAXIMUM_INTERVAL = "for a maximum of", INTERVAL;
TIME = WHILE;
WHILE = "while", SIMPLE_TRIGGER;
```

5.3.2 Simple Trigger(implies)

As all *implies* Kernel Patterns within the Simple Trigger Category contain a form of "P implies Q", the first step is to find a clear structure for the implication. This is done via the *IMPLIES* rule. It encapsulates a *SIMPLE_TRIGGER* with "If" and "then", complying to the *If P then Q* structure found in the example sentences. The *Q* within *If P then Q* is represented by the *SIMPLE_ACTION* rule. The following sections introduce the unique attributes of each Kernel Pattern and explain how these are integrated into the grammatical structure.

P_implies_finally_globally_Q_B

P_implies_finally_globally_Q_B introduces *finally* and *globally* as temporal constraints. *Finally* consists of the words "within 10 seconds". As time Intervals are required for many patterns, the *INTERVAL* rule defines, that a time interval consists of a number and a time unit. *Globally* is simply represented by the word "continuously" in all example sentences. The resulting parse tree, which can identify "If TRIGGER then ACTION continuously within X TIME_UNITS", is depicted in Figure 5.10.

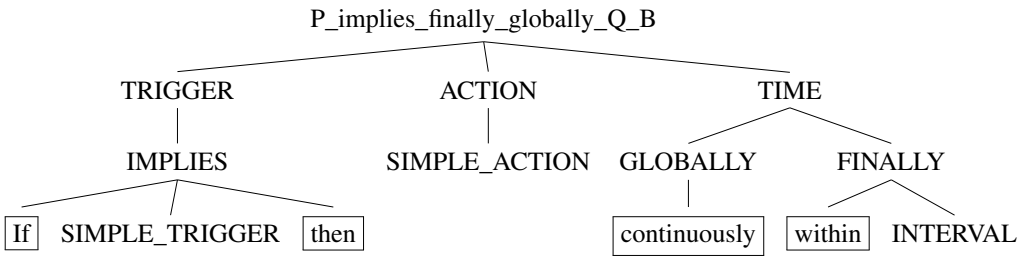

Figure 5.10: Parse tree for P_implies_finally_globally_Q_B

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", SIMPLE_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = GLOBALLY, FINALLY;
GLOBALLY = "continuously";
FINALLY = "within", INTERVAL;
```

P_implies_finally_Q_B

P_implies_finally_Q_B is grammatically equivalent to P_implies_finally_globally_Q_B apart from the fact, that *globally* is removed. The resulting parse tree, which can identify "If TRIGGER then ACTION within X TIME_UNITS", is depicted in Figure 5.11.

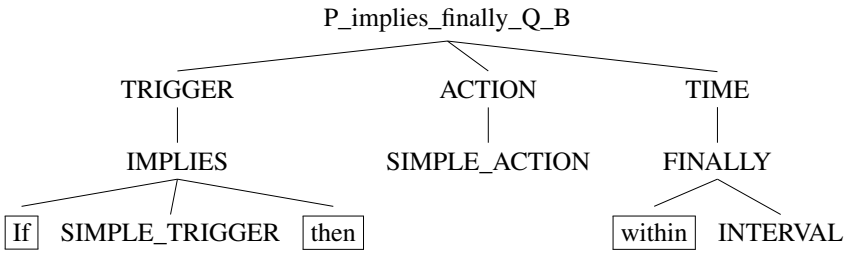

Figure 5.11: Parse tree for P_implies_finally_Q_B

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", SIMPLE_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = FINALLY;
FINALLY = "within", INTERVAL;
```

P_implies_globally_Q

P_implies_globally_Q is grammatically equivalent to P_implies_finally_globally_Q_B apart from the fact, that *finally* is removed. The resulting parse tree, which can identify "If TRIGGER then ACTION continuously", is depicted in Figure 5.12.

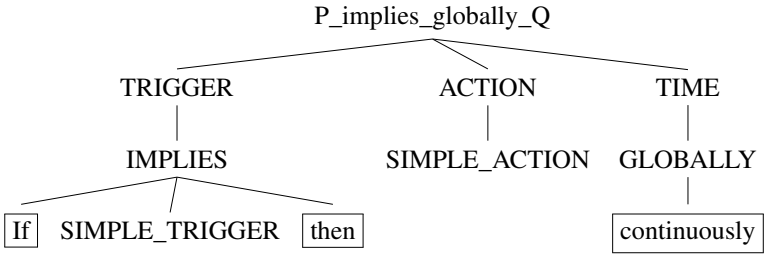

Figure 5.12: Parse tree for P_implies_globally_Q

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", SIMPLE_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = GLOBALLY;
GLOBALLY = "continuously";
```

P_implies_Q_atleast_X_steps_after_P

P_implies_Q_atleast_X_steps_after_P has an extra constraint which defines, that while P must be true, Q must be true at the same time. Thus another rule, *DUAL_TRIGGER* is required, which enables kernel patterns to have two consecutive triggers. The resulting implication is "If P and Q, then ...". This Kernel Pattern also adds another temporal constraint: *atleast X steps after* In the example sentence, this constraint is represented by "at least 1 second after TRIGGER and TRIGGER" Thus, the rule ATLEAST_AFTER consists of "at least INTERVAL after DUAL_TRIGGER". The resulting parse tree, which can identify "If TRIGGER and TRIGGER then ACTION at least INTERVAL after TRIGGER and TRIGGER", is depicted in Figure 5.13.

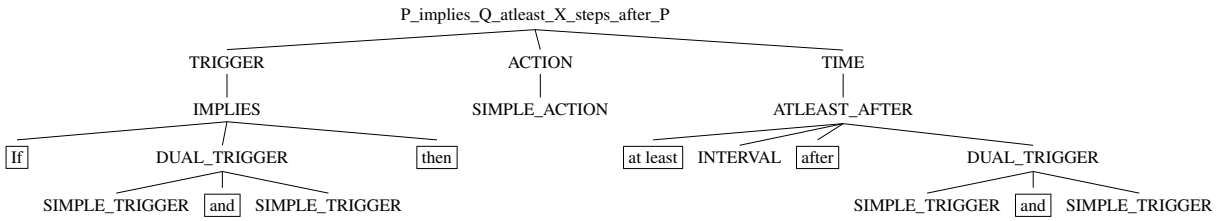

Figure 5.13: Parse tree for P_implies_Q_atleast_X_steps_after_P

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", DUAL_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = ATLEAST_AFTER;
ATLEAST_AFTER = "at least", INTERVAL, "after", DUAL_TRIGGER
DUAL_TRIGGER = SIMPLE_TRIGGER , "and", SIMPLE_TRIGGER;
```

P_implies_Q_during_X_steps

P_implies_Q_during_X_steps has the same implication as P_implies_Q_atleast_X_steps_after_P, in that it requires a DUAL_TRIGGER. This Kernel Pattern introduces *during* as a new temporal constraint. Within the examples, *during* is represented by "for 30 seconds", thus resulting in "for INTERVAL" as a rule. The resulting parse tree, which can identify "If TRIGGER and TRIGGER then ACTION during INTERVAL", is depicted in Figure 5.14.

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", DUAL_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = DURING;
DURING = "during the next", INTERVAL;
DUAL_TRIGGER = SIMPLE_TRIGGER , "and", SIMPLE_TRIGGER;
```

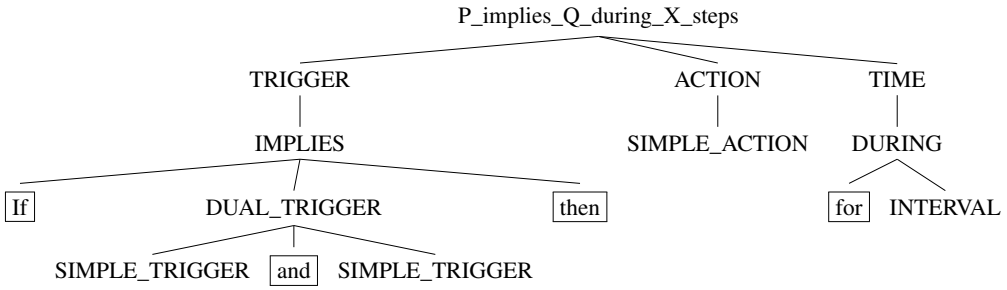

Figure 5.14: Parse tree for P_implies_Q_during_X_steps

P_implies_Q_during_next_X_steps

P_implies_Q_during_next_X_steps is grammatically equivalent to P_implies_Q_atleast_X_steps_after_P with the exception, that it does not require a *DUAL_TRIGGER*. The resulting parse tree, which can identify "If TRIGGER then ACTION during INTERVAL", is depicted in Figure 5.15.

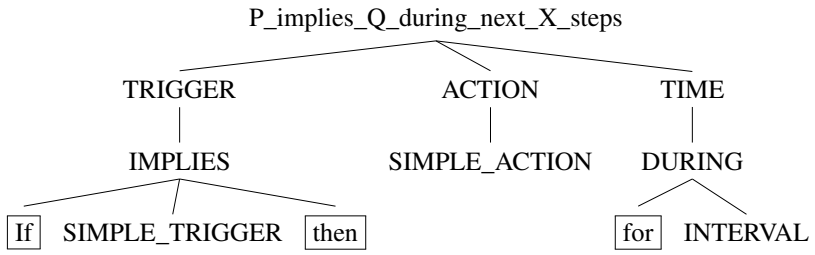

Figure 5.15: Parse tree for P_implies_Q_during_next_X_steps

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", SIMPLE_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = DURING;
DURING = "during the next", INTERVAL;
```

P_implies_Q_at_step_X_thereafter

P_implies_Q_at_step_X_thereafter introduces a new temporal constraint which results in the *EXACTLY_THEREAFTER* rule. This rule is constructed as "exactly INTERVAL thereafter". The resulting parse tree, which can identify "If TRIGGER then ACTION exactly INTERVAL thereafter", is depicted in Figure 5.16.

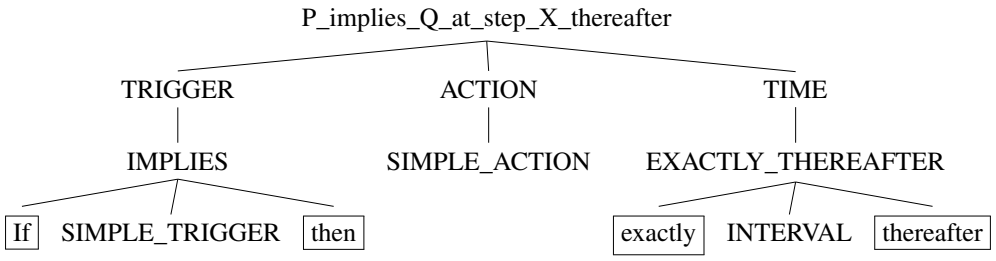

Figure 5.16: Parse tree for P_implies_Q_at_step_X_thereafter

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", SIMPLE_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = EXACTLY_THEREAFTER;
DURING = "exactly", INTERVAL, "thereafter";
```

P_implies_Q_X_steps_later

P_implies_Q_X_steps_later is grammatically equivalent to P_implies_Q_at_step_X_thereafter with the exception of it requiring a DUAL_TRIGGER. The resulting parse tree, which can identify "If TRIGGER and TRIGGER then ACTION exactly INTERVAL thereafter", is depicted in Figure 5.17.

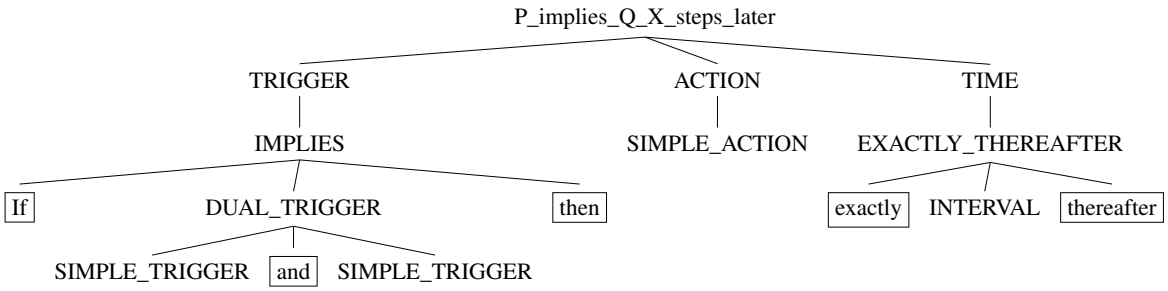

Figure 5.17: Parse tree for P_implies_Q_X_steps_later

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", DUAL_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = EXACTLY_THEREAFTER;
DURING = "exactly", INTERVAL, "thereafter";
DUAL_TRIGGER = SIMPLE_TRIGGER , "and", SIMPLE_TRIGGER;
```

5.3.3 Simple Trigger(triggers)

Grammatically, The *triggers* Kernel Patterns use the same *IMPLIES* structure as *implies* Kernel patterns. The *triggers* Kernel Patterns, however, all require a *DUAL_TRIGGER*, whereas the trigger P and the action Q are inserted for the two triggers. This determines, that the trigger P and the action Q must occur simultaneously.

P_triggers_Q_unless_S

In addition to the constraint mentioned in the previous section, *P_triggers_Q_unless_S* introduces the *UNLESS* rule, which is constructed as "until TRIGGER". The resulting parse tree, which can identify "If TRIGGER and TRIGGER then ACTION until TRIGGER", is depicted in Figure 5.18.

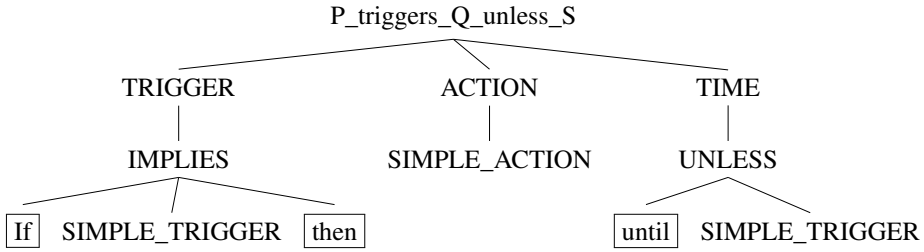

Figure 5.18: Parse tree for P_triggers_Q_unless_S

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", SIMPLE_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = UNLESS;
UNLESS = "until" SIMPLE_TRIGGER;
```

P_triggers_Q_unless_S_within_B

P_triggers_Q_unless_S_within_B extends P_triggers_Q_unless_S by a *FINALLY* rule after the *UNLESS* rule. The resulting parse tree, which can identify "If TRIGGER and TRIGGER then ACTION until TRIGGER within INTERVAL", is depicted in Figure 5.19.

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
IMPLIES = "If", SIMPLE_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = UNLESS, FINALLY;
UNLESS = "until" SIMPLE_TRIGGER;
FINALLY = "within", INTERVAL;
```

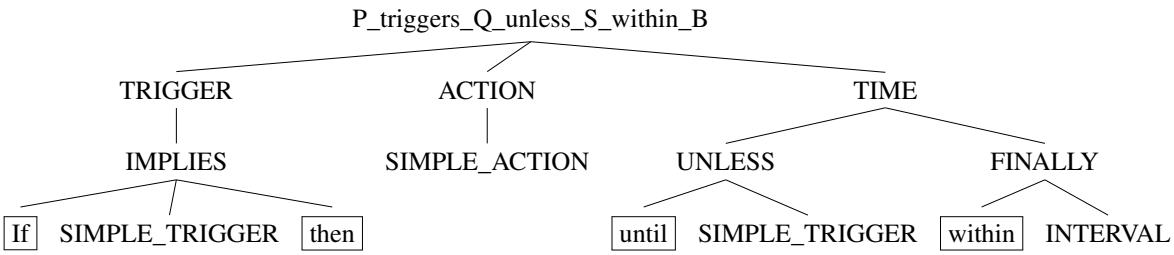

Figure 5.19: Parse tree for P_triggers_Q_unless_S_within_B

5.3.4 Temporal Trigger(stable implies)

The Kernel Patterns of this category expect the *TRIGGER* to be satisfied for a given period of time. As this cannot be represented by the current *IMPLIES* rule for Triggers, a new rule is required. This new rule, *STABLE*, must accept "If SIMPLE_TRIGGER for INTERVAL, then" and can be identified within the Kernel Patterns name, as all these Kernel Patterns' names contain "P_stable_X_steps". The parse tree is depicted in Figure 5.20

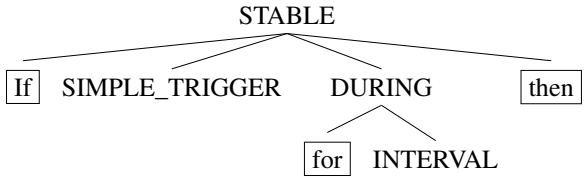

Figure 5.20: Parse tree for the STABLE rule.

This parse tree represents the following grammar rules:

```
TRIGGER = STABLE;
STABLE = "If", SIMPLE_TRIGGER, "for", INTERVAL, "then";
```

P_stable_X_steps_implies_afterwards_Q

P_stable_X_steps_implies_afterwards_Q simply implies the *Action Q*, if the *Trigger P* was stable for a given interval. Thus, it consists of the *STABLE* rule for the Trigger and a *SIMPLE_ACTION* rule for the Action. The resulting parse tree, which can identify "If TRIGGER for INTERVAL, then ACTION.", is depicted in Figure 5.21.

This parse tree represents the following grammar rules:

```
TRIGGER = STABLE;
STABLE = "If", SIMPLE_TRIGGER, "for", INTERVAL, "then";
ACTION = SIMPLE_ACTION;
```

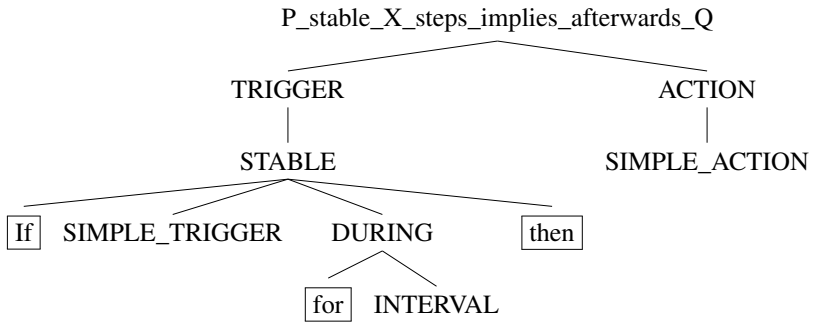

Figure 5.21: Parse tree for P_stable_X_steps_implies_afterwards_Q

P_stable_X_steps_implies_finally_Q_B

P_stable_X_steps_implies_finally_Q_B simply implies the *Action Q*, if the *Trigger P* was stable for a given interval. Thus, it consists of the *STABLE* rule for the Trigger and a *SIMPLE_ACTION* rule for the Action. The resulting parse tree, which can identify "If TRIGGER for INTERVAL, then ACTION within INTERVAL.", is depicted in Figure 5.22.

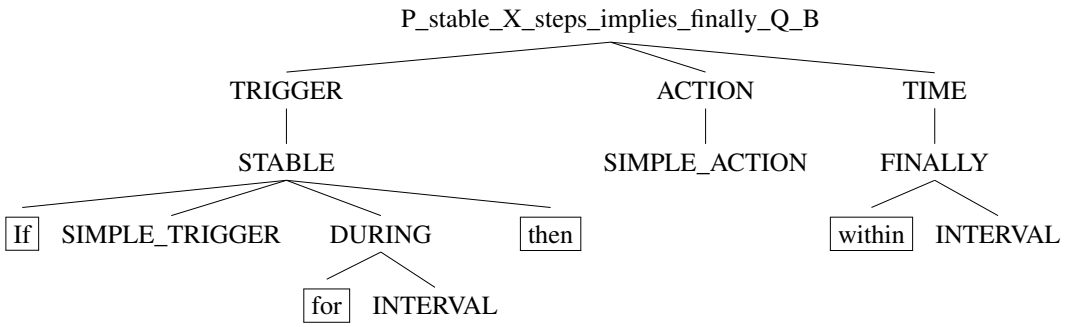

Figure 5.22: Parse tree for P_stable_X_steps_implies_finally_Q_B

This parse tree represents the following grammar rules:

```
TRIGGER = STABLE;
STABLE = "If", SIMPLE_TRIGGER, "for", INTERVAL, "then";
ACTION = SIMPLE_ACTION;
TIME = FINALLY;
FINALLY = "within", INTERVAL;
```

P_stable_X_steps_implies_Q_within_Y_steps_unless_S

P_stable_X_steps_implies_finally_Q_B introduces "within_Y_steps" in its name. Indeed this is an alternative representation of the *FINALLY* rule. The *TIME* component consists of the known *FINALLY* and *UNLESS* rules. The resulting parse tree, which can identify "If TRIGGER for INTERVAL, then ACTION within INTERVAL unless TRIGGER.", is depicted in Figure 5.23.

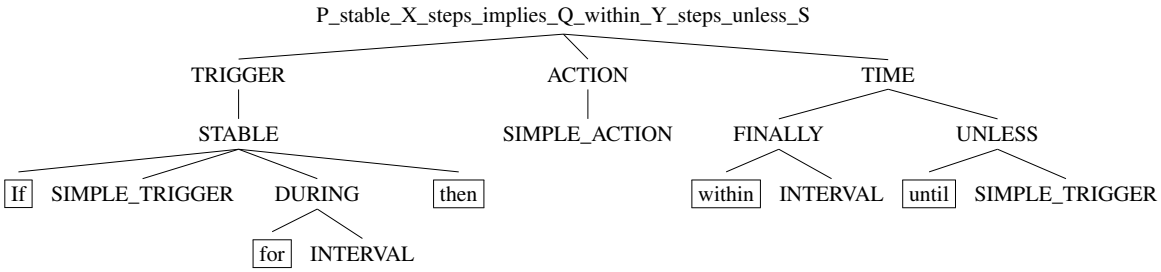

Figure 5.23: Parse tree for P_stable_X_steps_implies_Q_within_Y_steps_unless_S

This parse tree represents the following grammar rules:

```
TRIGGER = STABLE;
STABLE = "If", SIMPLE_TRIGGER, "for", INTERVAL, "then";
ACTION = SIMPLE_ACTION;
TIME = FINALLY, UNLESS;
FINALLY = "within", INTERVAL;
UNLESS = "until", SIMPLE_TRIGGER;
```

P_stable_X_steps_implies_Q_stable_Y_steps_within_B_steps

P_stable_X_steps_implies_Q_stable_Y_steps_within_B_steps extends the *P_stable_X_steps_implies_finally_Q_B* kernel pattern by the property, that the action must hold for Y steps. The resulting parse tree, which can identify "If TRIGGER for INTERVAL, then ACTION for INTERVAL within INTERVAL.", is depicted in Figure 5.24.

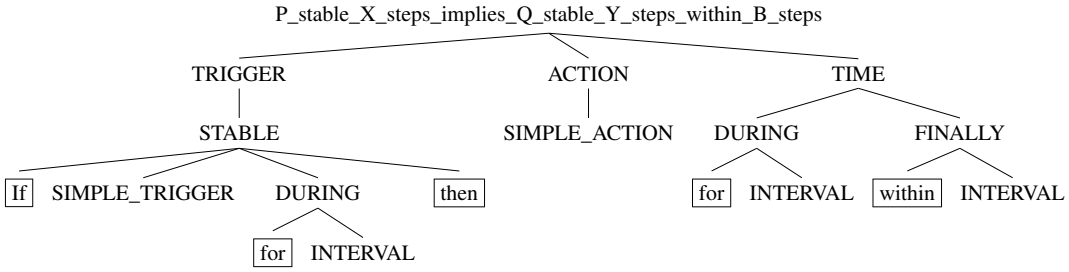

Figure 5.24: Parse tree for P_stable_X_steps_implies_Q_stable_Y_steps_within_B_steps

This parse tree represents the following grammar rules:

```
TRIGGER = STABLE;
STABLE = "If", SIMPLE_TRIGGER, "for", INTERVAL, "then";
ACTION = SIMPLE_ACTION;
TIME = DURING, FINALLY;
DURING = "during the next", INTERVAL;
FINALLY = "within", INTERVAL;
```

P_stable_X_steps_implies_globally_Q_within_Y_steps

P_stable_X_steps_implies_globally_Q_within_Y_steps extends *P_stable_X_steps_implies_finally_Q_B* by the *GLOBALLY* rule. The resulting parse tree, which can identify "If TRIGGER for INTERVAL, then ACTION continuously within INTERVAL.", is depicted in Figure 5.25.

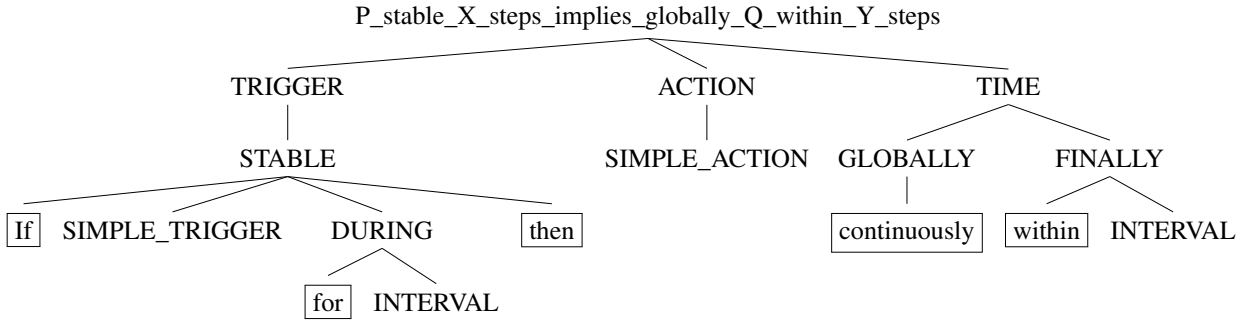

Figure 5.25: Parse tree for P_stable_X_steps_implies_globally_Q_within_Y_steps

This parse tree represents the following grammar rules:

```
TRIGGER = STABLE;
STABLE = "If", SIMPLE_TRIGGER, "for", INTERVAL, "then";
ACTION = SIMPLE_ACTION;
TIME = GLOBALLY, FINALLY;
GLOBALLY = "continuously";
FINALLY = "within", INTERVAL;
```

P_stable_X_steps_implies_Q_stable_Y_steps_B_steps_thereafter

P_stable_X_steps_implies_Q_stable_Y_steps_B_steps_thereafter requires the *DURING* and *EXACTLY_THEREAFTER* rules within its *TIME* component. The resulting parse tree, which can identify "If TRIGGER for INTERVAL, then ACTION for INTERVAL exactly INTERVAL thereafter.", is depicted in Figures 5.26 and 5.27.

These parse trees represent the following grammar rules:

```
TRIGGER = STABLE;
STABLE = "If", SIMPLE_TRIGGER, "for", INTERVAL, "then";
ACTION = SIMPLE_ACTION;
TIME = DURING, EXACTLY_THEREAFTER;
DURING = "during the next", INTERVAL;
EXACTLY_THEREAFTER = "exactly", INTERVAL, "thereafter";
```

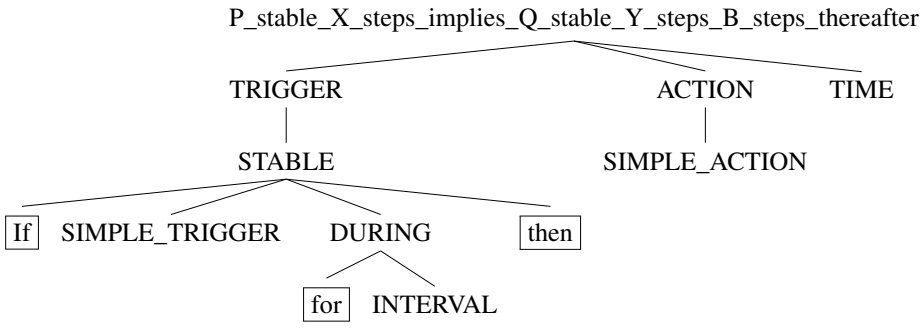

Figure 5.26: Parse tree for P_stable_X_steps_implies_Q_stable_Y_steps_B_steps_thereafter

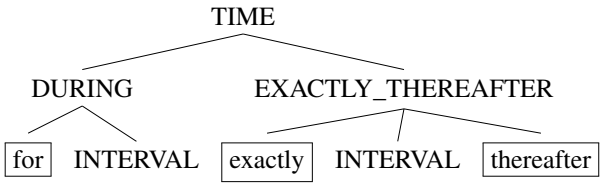

Figure 5.27: TIME subtree for P_stable_X_steps_implies_Q_stable_Y_steps_B_steps_thereafter

5.3.5 Temporal Trigger(stable triggers releasing)

The *Temporal Trigger(stable triggers releasing)* category consists of a single Kernel Pattern, which extends *P_stable_X_steps_implies_Q_within_Y_steps_unless_S* by the property, that the Action must be "released" by a second Action.

P_stable_X_steps_triggers_S_releasing_Q_within_Y_steps

P_stable_X_steps_triggers_S_releasing_Q_within_Y_steps extends *P_stable_X_steps_implies_Q_within_Y_steps_unless_S* by a DUAL_TRIGGER within the *UNLESS* rule, whereas the two Triggers within the *UNLESS* rule are the predicates *Q* and *S*. The resulting parse tree, which can identify "If TRIGGER for INTERVAL, then ACTION within INVERVAL until TRIGGER and TRIGGER.", is depicted in Figures 5.28 and 5.29.

These parse trees represent the following grammar rules:

```
TRIGGER = STABLE;
STABLE = "If", SIMPLE_TRIGGER, "for", INTERVAL, "then";
ACTION = SIMPLE_ACTION;
TIME = FINALLY, UNLESS;
FINALLY = "within", INTERVAL;
UNLESS = "until", DUAL_TRIGGER;
DUAL_TRIGGER = SIMPLE_TRIGGER, "and", SIMPLE_TRIGGER;
```

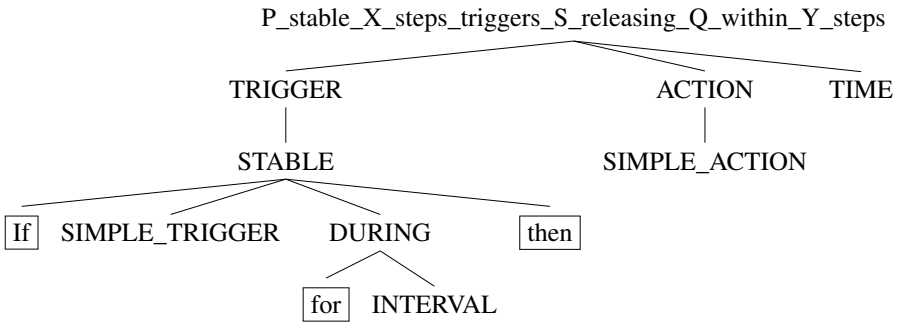

Figure 5.28: Parse tree for P_stable_X_steps_triggers_S_releasing_Q_within_Y_steps

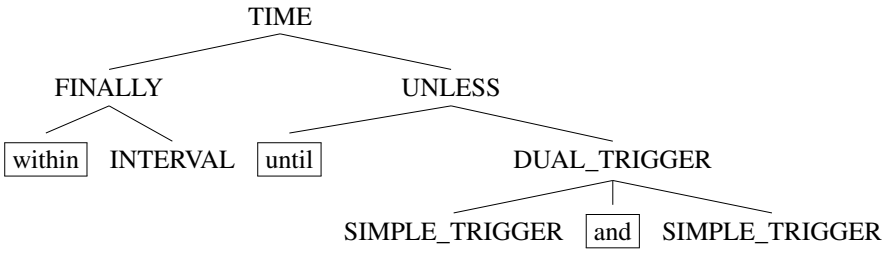

Figure 5.29: TIME subtree for P_stable_X_steps_triggers_S_releasing_Q_within_Y_steps

5.3.6 Temporal Trigger(triggering stable implies)

P_triggering_Q_stable_X_steps_implies_S_within_Y_steps_if_stable_T

P_triggering_Q_stable_X_steps_implies_S_within_Y_steps_if_stable_T combines *DURING*, *AFTER_WHICH* and *UNLESS* with a *DUAL_TRIGGER* to constitute its *TIME* component. This is a complex Kernel Pattern which has an additional constraint for the *DUAL_TRIGGER* within the *UNLESS* rule. For the grammar structure, this constraint is not relevant at this step.

The resulting parse tree, which can identify "If TRIGGER, then ACTION for INTERVAL, after which ACTION for a maximum of INTERVAL or until TRIGGER.", is depicted in Figures 5.30 and 5.31.

These parse trees represent the following grammar rules:

```
TRIGGER = IMPLIES;
STABLE = "If", SIMPLE_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = DURING, AFTER_WHICH_OR;
DURING = "for", INTERVAL;
AFTER_WHICH_OR = AFTER_WHICH, MAXIMUM_INTERVAL, "or", UNLESS;
AFTER_WHICH = "after which", SIMPLE_TRIGGER;
MAXIMUM_INTERVAL = "for a maximum of", INTERVAL;
UNLESS = "until", SIMPLE_TRIGGER;
```

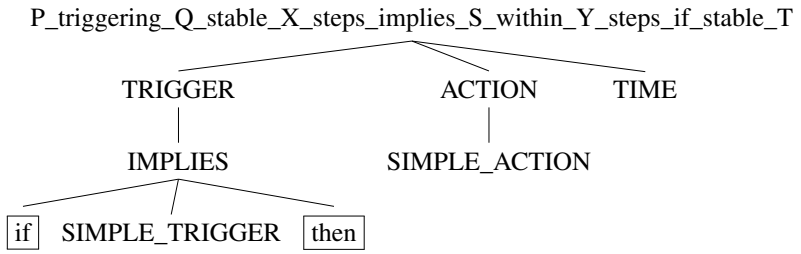

Figure 5.30: Parse tree for P_triggering_Q_stable_X_steps_implies_S_within_Y_steps_if_stable_T

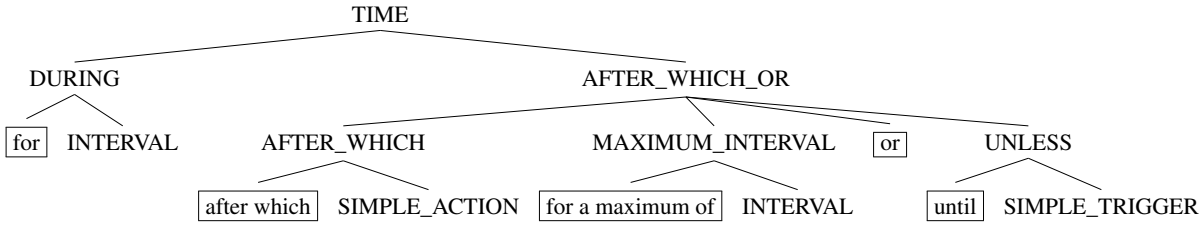

Figure 5.31: TIME subtree for P_triggering_Q_stable_X_steps_implies_S_within_Y_steps_if_stable_T

5.3.7 Temporal Trigger(triggering within implies)

P_triggering_Q_within_X_steps_implies_S_within_Y_steps

P_triggering_Q_within_X_steps_implies_S_within_Y_steps

The resulting parse tree, which can identify "If TRIGGER, then ACTION within INTERVAL, after which ACTION within INTERVAL." , is depicted in Figure 5.32.

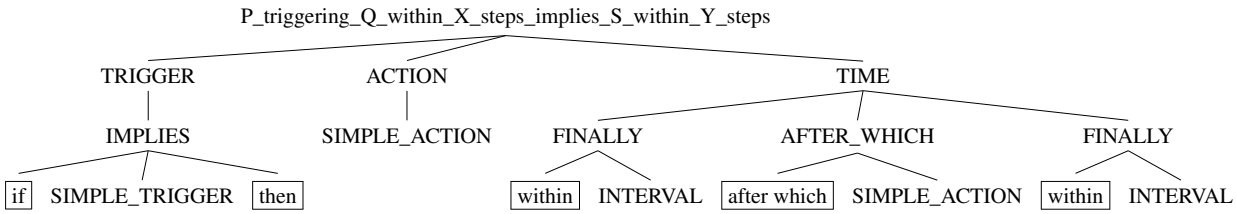

Figure 5.32: Parse tree for P_triggering_Q_within_X_steps_implies_S_within_Y_steps

This parse tree represents the following grammar rules:

```
TRIGGER = IMPLIES;
STABLE = "If", SIMPLE_TRIGGER, "then";
ACTION = SIMPLE_ACTION;
TIME = FINALLY, AFTER_WHICH, FINALLY;
FINALLY = "within", INTERVAL;
AFTER_WHICH = "after which", SIMPLE_TRIGGER;
```

5.3.8 No Trigger

No Trigger Kernel Patterns do not contain any *TRIGGER* as part of its structure. The resulting maximum structure of a *No Trigger* Kernel Pattern is depicted in Figure 5.33.

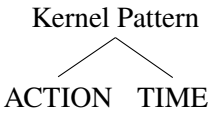

Figure 5.33: Abstract parse tree for Kernel Patterns.

P

P merely consists of an ACTION. The resulting parse tree, which can identify "ACTION.", is depicted in Figure 5.34.

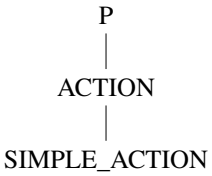

Figure 5.34: Parse tree for P

This parse tree represents the following grammar rules:

```
ACTION = SIMPLE_ACTION;
```

finally_P_B

finally_P_B extends *P* by a FINALLY rule. The resulting parse tree, which can identify "ACTION within INTERVAL.", is depicted in Figure 5.35.

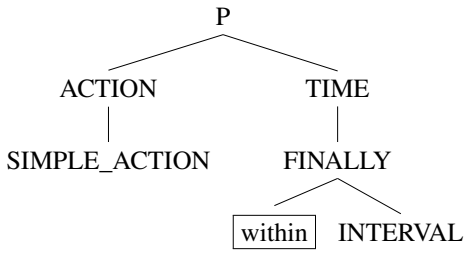

Figure 5.35: Parse tree for finally_P_B

This parse tree represents the following grammar rules:

```
ACTION = SIMPLE_ACTION;
TIME = FINALLY;
FINALLY = "within", INTERVAL;
```

finally_globally_P_B

finally_globally_P_B extends *finally_globally_P_B* by a *GLOBALLY* rule. The resulting parse tree, which can identify "ACTION continuously within INTERVAL.", is depicted in Figure 5.36.

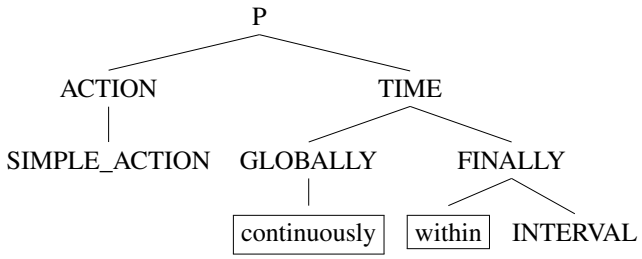

Figure 5.36: Parse tree for finally_globally_P_B

This parse tree represents the following grammar rules:

```
ACTION = SIMPLE_ACTION;
TIME = GLOBALLY, FINALLY;
GLOBALLY = "continuously";
FINALLY = "within", INTERVAL;
```

5.3.9 Ordering

Ordering Kernel Patterns are grammatically similar to **Invariant** Kernel Patterns in section 5.3.1, as both have their *Trigger* within a temporal constraint after the *Action*.

Q_onlyafter_P

Q_onlyafter_P consists of a SIMPLE_ACTION and a *TIME* rule, which must evaluate to "only after" SIMPLE_TRIGGER. This rule is introduced with *ONLY_AFTER*. The resulting parse tree, which can identify "ACTION only after TRIGGER.", is depicted in Figure 5.37.

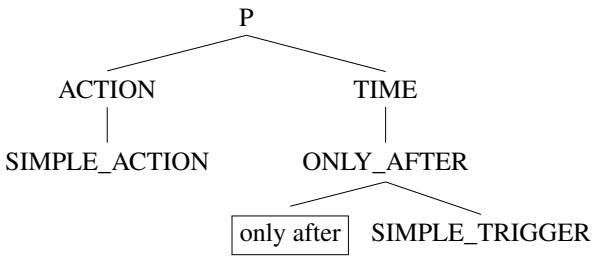

Figure 5.37: Parse tree for finally_globally_P_B

This parse tree represents the following grammar rules:

```
ACTION = SIMPLE_ACTION;
TIME = ONLY_AFTER;
FINALLY = "only after", SIMPLE_TRIGGER;
```

Q_notbefore_P

Q_notbefore_P requires that its Action is negated. Due to the versatility of negation within natural language, the grammar cannot check this constraint. It is therefore assumed, that the ACTION is negated. The TIME rule *BEFORE* can be used to parse for "before TRIGGER", which completes the Kernel Pattern given the aforementioned assumption. The resulting parse tree, which can identify "ACTION before TRIGGER.", is depicted in Figure 5.38.

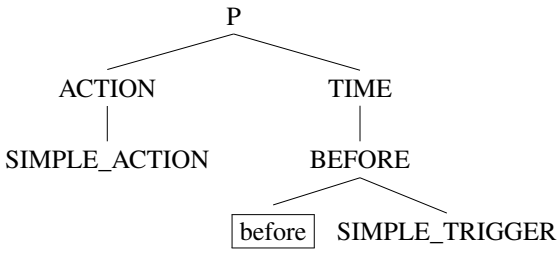

Figure 5.38: Parse tree for Q_notbefore_P

This parse tree represents the following grammar rules:

```
ACTION = SIMPLE_ACTION;
TIME = BEFORE;
FINALLY = "before", SIMPLE_TRIGGER;
```

5.4 Constructing the Grammar

By concatenating the grammar rules of all kernel patterns in section 5.3, a grammar can be constructed. In order to construct an extendable, readable grammar, abstraction is required. The combination of the rules *Q_while_P* and *Q_while_P_B* for example, results in a single ruleset, whereby the non-terminal symbol *MAXIMUM_INTERVAL* becomes optional, as can be seen in Figure 5.39.

```
PATTERN = KERNEL_PATTERN
KERNEL_PATTERN = ACTION, TIME;
ACTION = SIMPLE_ACTION, [MAXIMUM_INTERVAL];
MAXIMUM_INTERVAL = "for a maximum of", INTERVAL;
TIME = WHILE;
WHILE = "while", SIMPLE_TRIGGER;
```

Figure 5.39: Grammar for the kernel patterns Q_while_P and Q_while_P_B

By adding the *ACTIVATION_MODE* and *START-UP* phase to the *PATTERN* and adding all *KERNEL_PATTERN* rules one after another, the complete grammar comes into existance. This grammar is very primitive and is extended in section 5.5, so that it can be used for the prototype within this thesis.

Grammar for the example requirements

```
1  PATTERN = [ACTIVATION_MODE], KERNEL_PATTERN, [START-UP];
2  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Activation Mode
3  ACTIVATION_MODE = INITIAL_AM
4                  | FIRST_AM;
5  INITIAL_AM = "initially" | "Directly after start-up" | "After start-up";
6  FIRST_AM = "For one occurence" | "For the first occurence" | "For the first time";
7  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Start-Up Phase
8  START-UP  = AFTER_REACHING_R
9            | AFTER_N_STEPS;
10 AFTER_REACHING_R = "After", INTERVAL;
11 AFTER_N_STEPS = "After", PREDICATE;
12 // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Kernel Pattern
13 KERNEL_PATTERN =
14     [TRIGGER], ACTION, [TIME];
15 // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Trigger
16 TRIGGER =
17     IMPLIES | STABLE;
18 IMPLIES =
19     IF, SIMPLE_TRIGGER, THEN
20     | IF, DUAL_TRIGGER, THEN;
21 DUAL_TRIGGER =
22     SIMPLE_TRIGGER, AND, SIMPLE_TRIGGER;
23 SIMPLE_TRIGGER =
24     PREDICATE;
25 STABLE =
26     IF, SIMPLE_TRIGGER, DURING, THEN;
27 PREDICATE =
28     UNINTERPRETED_STRING; // A common string in the context of computer science.
29 IF =
```

```
    "If" | "When";
THEN =
    ", then" | ",";



// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Action
ACTION =
    SIMPLE_ACTION;
SIMPLE_ACTION =
    PREDICATE;



// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Time
TIME =
    WHILE | MAXIMUM_INTERVAL | ATLEAST_AFTER | DURING | ONLY_AFTER | BEFORE
    | [ (GLOBALLY | DURING | UNLESS | FINALLY, AFTER_WHICH) ], FINALLY | GLOBALLY
    | [DURING], EXACTLY_THEREAFTER | [FINALLY], UNLESS | DURING, AFTER_WHICH_OR;
WHILE =
    "while", SIMPLE_TRIGGER;
MAXIMUM_INTERVAL =
    MAXIMUM, INTERVAL;
GLOBALLY =
    "continuously";
FINALLY =
    WITHIN, INTERVAL;
ATLEAST_AFTER =
    ATLEAST, INTERVAL, AFTER, DUAL_TRIGGER;
DURING =
    DURING_WORD, INTERVAL;
EXACTLY_THEREAFTER =
    EXACTLY, INTERVAL, THEREAFTER;
UNLESS =
    UNLESS_WORD, (SIMPLE_TRIGGER | DUAL_TRIGGER);
AFTER_WHICH_OR =
    AFTER_WHICH, MAXIMUM_INTERVAL, OR, UNLESS;
AFTER_WHICH =
    AFTER_WHICH_WORD, SIMPLE_TRIGGER;
ONLY_AFTER =
    "only after", SIMPLE_TRIGGER;
BEFORE =
    "before", SIMPLE_TRIGGER;



INTERVAL =
    NUMBER, TIME_UNIT;
NUMBER =
    {DIGIT}-;
DIGIT =
    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
TIME_UNIT =
    {"nanosecond" | "millisecond" | "second" |
    "minute" | "hour" | "day" | "step"}, ["s"];

MAXIMUM =
    "for a maximum of";
ATLEAST =
    "at least";
AFTER =
    "after";
AND =
    "and";
OR =
    "or";
```

```
DURING_WORD =
    "for" | "for the next" | "during the next";
EXACTLY =
    "exactly" | "precisely";
THEREAFTER =
    "thereafter" | "later";
UNLESS_WORD =
    "until";
AFTER_WHICH_WORD =
    "after which";
```

5.5 Grammar Validation

The grammar in section 5.4 covers all *kernel patterns* along with the *start-up Phases* and *activation modes* presented in chapter 4. This grammar, however, was built upon the fictional example requirements from section 4.2. The required extension of the grammar, to comply with variable natural language specifications, is displayed in section 5.5.6. As this grammar is to be used to parse real world requirements, this section analyzes the structure of example requirements by end consumers of BTC EmbeddedSystems AG, which are not named due to confidentiality.

The common structures of five requirements documents are analyzed within the following sections.

5.5.1 Implications

The *implication*, **If TRIGGER, then ACTION**, is the most common structure within the grammar and also the most commonly used structure within requirements. Within the requirements documents, **ACTION when TRIGGER**, **TRIGGER triggers ACTION**, **ACTION when TRIGGER**, **Once TRIGGER, then ACTION** and **ACTION if TRIGGER** are used to represent implications.

While most of these formulations are used very rarely, **ACTION when TRIGGER** is used very commonly by different customers. Thus, it should be included as an alternative rule within the grammar. As the *Trigger* component is present behind the *Action* component, a new type of *Trigger* component is required. This is denoted as the *Post-Trigger*.

5.5.2 Requirement complexity

Many requirements within the documents grouped several different *Triggers* or several different *Actions* within a single requirement. As the BTC EmbeddedSpecifier can produce Observers which "test" the system, one should attempt to cover a single case with one requirement. This is also mentioned as a best-practice by Pohl and Rupp, despite the fact, that requirements documents may become very large and hard to navigate [PR11].

5.5.3 Temporal constraints

The documents include heavy use of *until* and *within* in the requirements. These keywords are used exactly as they are used within the fictional examples, thus not posing any problem at all. Other temporal constraints are not used within the example documents. Further conditions, which are attached to the *Triggers* and *Actions* indicate that the requirement should be split into multiple requirements, as these conditions can be checked in another requirement far more easily.

5.5.4 The Kernel Pattern P

The Kernel Pattern **P** can be found very often within most requirement documents. The content of these requirements are often similar to these examples:

- Signal A is transferred via the global Bus system.

- The emergency light blinks in a frequency of 2 Hertz.
- The fuel display shows the amount of fuel within the tank in liters.

These requirements cannot be tested in this state. Using *Temporal Trigger Kernel Patterns*, features such as the 2 Hertz frequency can be tested. Other requirements would have to be tested in more detail.

5.5.5 Start-Up Phase and Activation Mode

Rarely, a sentence is formulated like **After TRIGGER, If TRIGGER, then ACTION**. This corresponds to an implication as it is implemented within the grammar, in addition of an **Activation Mode**. No start-up phases were used within the example documents. Instead, the start-up Phase was often described in advance before the requirements were noted.

5.5.6 Extending the Grammar

The grammar in section 5.4 covers the exact structures found within the example requirements in section 5.3. In order to also accept the example requirements from customers of BTC EmbeddedSystems, as well as many unknown natural language specifications and their structures, the grammar must offer freedom in both linguistic structure and wording. Extending the freedom in wording is done by adding synonyms to non-terminal symbols. Freedom in structure is a more complex task. Implications, which are resembled by the *IMPLIES* non-terminal symbol, are the most common structure within the kernel patterns. By allowing *TRIGGER*, *ACTION* and *TIME* to occur in any reasonable order, the linguistic structure of implications can adopt several forms. This structural freedom, which is displayed in Figure 5.40 is applied to nearly all kernel patterns, allowing the extended grammar to accept a multiple of the previously defined grammar.

```
KERNEL_PATTERN = TRIGGER, ACTION, TIME
               | ACTION, TIME, TRIGGER
               | TIME, ACTION, TRIGGER
```

Figure 5.40: Possible orders for Trigger, Action, and Time

The extended grammar, which is implemented in the prototype, is depicted within the appendix in section 10.

5.5.7 Freedom of Natural Language

Natural language offers superior freedom in its structure, as well as in its words. A single requirement can have a vast amount of representations in natural language. A natural language requirement can be represented by multiple structurally different sentences, while preserving most of the words. While the grammar can be extended to accept multiple synonyms to replace keywords, such as *while* or *if*, covering all synonyms or all structures, which the natural language requirement can assume is not achievable. Even if it was possible to cover 100 percent of the possible linguistic representations,

an unambiguous mapping to a kernel pattern and its macros would remain impossible due to the ambiguity of natural language.

5.5.8 Conclusion

While the grammar can detect all Kernel Patterns, it cannot identify all real world requirements. While many requirements would only require small changes, such as **When Trigger, then Action** would be changed to **If Trigger, then Action**, other requirements pose a problem. These are usually requirements, which could also not be unambiguously tested. By reducing requirements to a single, testable aspect, the formulation of the requirement in a grammar compliant language becomes is significantly simplified. Many requirements exist, which are not accepted by the grammar even after reduction. The grammar can be extended to accept a greater spectrum of requirements, as long as these are not ambiguous. This property can be used to adapt the grammar to different customer's needs.

6 Meta-Model

The grammar defined in section 5 techniques described in the previous sections can be used to successfully identify patterns. By confining oneself to using a grammar, the pattern recognition must be made directly while parsing the input. This approach is very difficult to extend and it is directly bound to the linguistic grammar of the input sentence.

6.1 Meta-Model of a Pattern

The prototypes parser can transfer the natural language specification into an abstract data type. This abstract data type must be able to represent instances of every kernel pattern in the BTC Embedded-Validator Pattern Library Release 3.7 [BTC12] As a pattern is an instance of a kernel pattern with a start-up phase and an activation mode, every parsed requirement must be representable as an instance of the meta-model. Figure 6.1 represents this meta-model while taking into account, that in natural language, one can omit the start-up phase and activation mode. In this case, a default behaviour is implemented for the activation mode and start-up phase. Meta-Models are extendable, allowing for an easy addition of new kernel patterns.

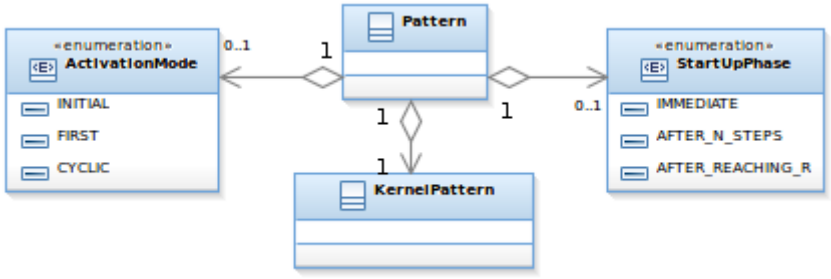

Figure 6.1: Meta-model for patterns.

6.2 Meta-Model of a Kernel Pattern

The models of kernel patterns differ from each other by the number of predicates and intervals, as well as the various keywords they contain. Thus, the meta-model must be able to hold multiple keywords and an arbitrary number of intervals and predicates. The keywords must containt all keywords contained in the common structures of the patterns in section 4. Also, keywords can occur multiple times in a single part of a kernel pattern, such as the *Time* component. Reliable identification of kernel patterns is not possible solely based on the quantity of these three components. By preserving some of the natural language specifications structure, namely the action, trigger and the temporal constraint (hereafter denoted as time), the pattern recognition becomes much preciser. The parser can aggregate the trigger, action and time while parsing, thus creating a stuctured instance of the meta-model in Figure 6.2. As is visible in this figure, an *action* merely contains predicates, whereas a *trigger* and a *time* can additionally comprise intervals and keywords.

Figure 6.3 depicts an instance of the simple *finally_P_B* kernel pattern. The pattern has one *predicate* as its *action* component. Its *time* component consists of one *interval* with one *FINALLY* keyword.

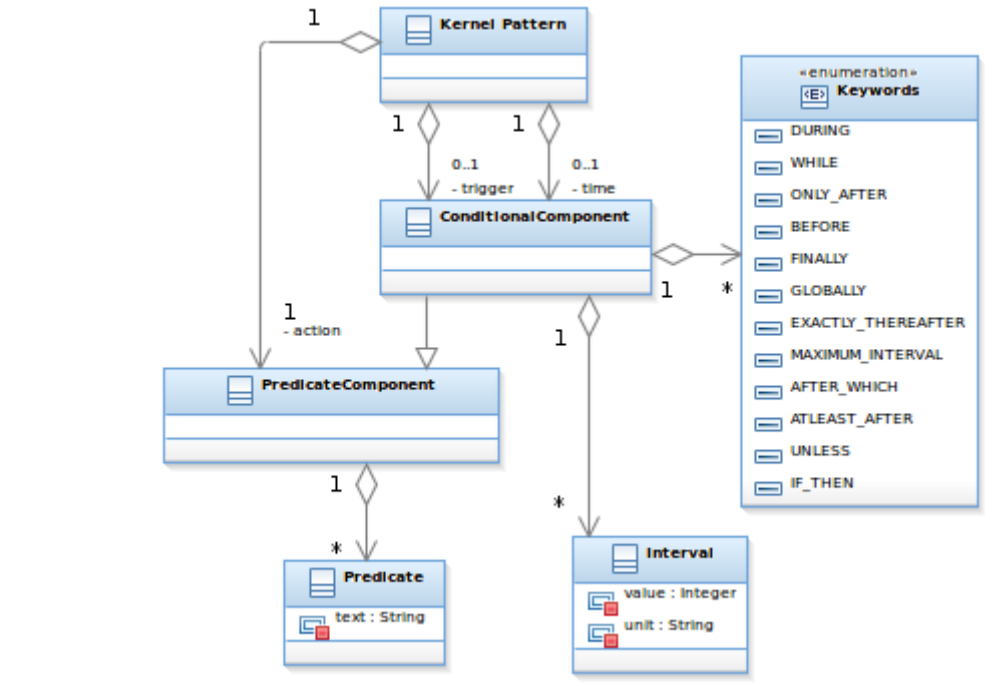

Figure 6.2: Meta-model for all kernel patterns.

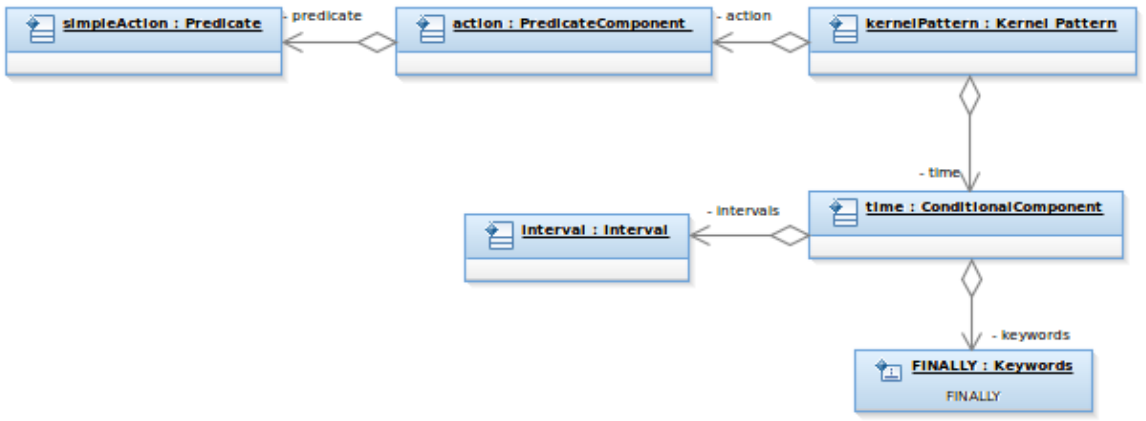

Figure 6.3: Model for finally_P_B.

Figure 6.4 displays an instance of the complex *P_triggering_Q_within_X_steps_implies_S_within_Y_steps* pattern. The pattern has one *predicate* as its *action* component. Its *trigger* component consists of one *predicate* and one *interval* with one *IF_THEN* and one *DURING* keywords. Its *time* component consists of one *predicate* and two *intervals* with one *AFTER_WHICH* aswell as two *FINALLY* keywords.

A complete collection of models for all kernel patterns is in the attachment in chapter 10.

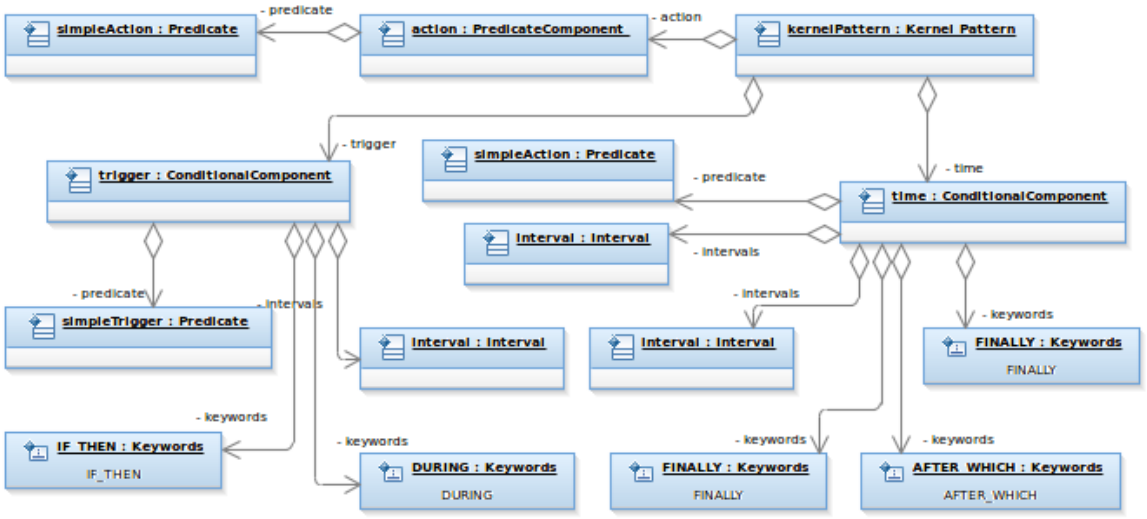

Figure 6.4: Model for P_triggering_Q_within_X_steps_implies_S_within_Y_steps.

7 Prototype

This chapter describes a prototype, which accepts a natural language specification as input and, provided the input conforms to the grammar defined in chapter 5, identifies applicable patterns from BTC EmbeddedSystems' BTC EmbeddedValidator Pattern Library Release 3.7 [BTC12] by means of the meta-model from section 6. The prototype is to be showcased to customers of BTC EmbeddedSystems to demonstrate the possibilities of automated pattern recognition. First the requirements for the prototype are established in section 7.1. The concept and approach , which are implemented in the prototype are introduced in section 7.2. Lastly, section 7.3 depicts the finished prototype. All UML models displayed within this chapter comply to the UML 2.4.1 superstructure [Obj13].

7.1 Requirements

The requirements for the prototype are classified in the three categories "constraints", "functional requirements" and "quality requirements", as defined by Pohl and Rupp [PR11]. Constraints in section 7.1.1 represent requirements, which must be satisfied due to limitations given by stakeholders, the environment of the software, or similar sources. The functional requirements in section 7.1.2 describe the functionality, which is to be implemented by the prototype. Quality requirements in section 7.1.3 describe non-functional properties, such as efficiency, maintainability or user-friendliness, which the prototype must satisfy. The key words "must", "must not", "required", "shall", "shall not", "should", "should", "recommended", "may", and "optional" in this document are to be interpreted as described in RFC 2119 [Bra97].

7.1.1 Constraints

The following constraints describe limits given by sources like stakeholders or the softwares environment, which the prototype must comply to. For this prototype, the constraints originate from the stakeholders at BTC EmbeddedSystems AG and the environment of the BTC EmbeddedSpecifier.

C.1) *The prototype must use the grammar specified in chapter 5, in order to analyze its input.* By defining a formal language using a grammar, the language can be parsed easily. A clearly defined grammar can be read by many computer scientists and is easily extendable.

C.2) *The prototype must use the meta-model from section 6 for pattern recognition.* Meta-models can easily be extended. This enables BTC EmbeddedSystems to incorporate new kernel patterns, which may be requested by customers.

C.3) *The meta-model used for pattern recognition must be able to represent all kernel patterns specified in the BTC EmbeddedValidator Pattern Library Release 3.7 [BTC12].* The Pattern Library contains all kernel patterns, which are used by the BTC EmbeddedSpecifier in chapter 3.

C.4) *The prototype should be implemented in Java.* This prototype can be used as a reference to implement a plugin for BTC EmbeddedSpecifier described in section 3, which is written in Java and based on the Eclipse project.

7.1.2 Functional Requirements

The prototype's purpose is to accept a natural language specification as input and recognize applicable patterns. In order to guarantee the fulfilment of this functionality, the following functional requirements are defined for the prototype:

F.1) *The prototype must accept a natural language specification as input.* The starting artifact in the BTC EmbeddedSpecifier is a natural language specification.

F.2) *The prototype must accept input up to at least 50 words in length.* Natural language specifications can become very lengthy, especially is taking complex kernel patterns into consideration. 50 words should cover most requirements.

F.3) *The prototype must recognize, if the natural language specifications structure corresponds to a pattern specified in the BTC EmbeddedValidator Pattern Library Release 3.7 [BTC12].* This is the intent and core functionality of the prototype. This requirement is bound to constraint C.2.

F.4) *The prototype must display, which kernel pattern the natural language specification corresponds to, if a match is found.* By precisely identifying the pattern, a plugin could automate the selection of this pattern within the BTC EmbeddedSpecifier.

F.5) *The prototype must display, which activation mode the natural language specification corresponds to, if a match is found.* By precisely identifying the pattern, a plugin could automate the selection of this pattern within the BTC EmbeddedSpecifier.

F.6) *The prototype must display, which start-up phase the natural language specification corresponds to, if a match is found.* By precisely identifying the pattern, a plugin could automate the selection of this pattern within the BTC EmbeddedSpecifier.

F.7) *The prototype must display an error, if the input doesn't conform to the grammar specified in chapter 5.* As stated by Shneiderman, a user interface must react to every action, even if this action is not succesful. [SP86]

F.8) *The prototype must display predicates and map them to the identified patterns macros.* If the macros are already identified, a plugin for the BTC EmbeddedSpecifier using this concept could automatically create a semi-formal representation of the requirement.

F.9) *The prototype must display possible alternative patterns, if no exact pattern match is found.* As the user must formalize all his requirements, the system ca help the user in case of a failed pattern recognition, by displaying patterns, which are similar to the natural language specification. The user can then update his natural language specification and attempt another pattern recognition.

7.1.3 Quality Requirements

This section defines the quality requirements for the prototype:

Q.1) *The pattern recognition should not take longer than 1 second.* While performance is not of high priority for the prototype, the pattern recognition must take place in a reasonable amount of time.

Q.2) *The meta-model for pattern recognition must be extendable by new kernel patterns.* BTC EmbeddedSystems' customers can request new kernel patterns. Thus the collection of kernel patterns is not static. In addition, patterns can be updated, as it has happened in Release 3.7 of the BTC EmbeddedValidator Pattern Library [BTC12].

Q.3) *The grammar for the natural language specification must be extendable.* BTC EmbeddedSystems' customers can request new kernel patterns. Thus the collection of kernel patterns is not static. In addition, patterns can be updated, as it has happened in Release 3.7 of the BTC EmbeddedValidator Pattern Library [BTC12].

Q.4) *The grammar must cover at least 3 structurally different natural language specifications for at least 5 patterns.* The grammar must be able to accept structurally different natural language specifications, as natural language requirements are formulated differently by different customers. Additionally, a system which only supports one static structure could not be easily adapted to different regional natural languages.

7.2 Concept and Approach

This section introduces the concept and approach used to fulfill the requirements defined in section 7.1. The prototype's goal is the translation of a natural language specification into a semi-formal specification in the form of a pattern. This workflow is depicted in Figure 7.1. After specifying the natural language specification, it must be parsed into an instance of the meta-model defined in chapter 6. This process is described in section 7.2.1. Section 7.2.2 describes the workflow required to recognize the pattern and its macros, thus resulting in a semi-formal requirement.

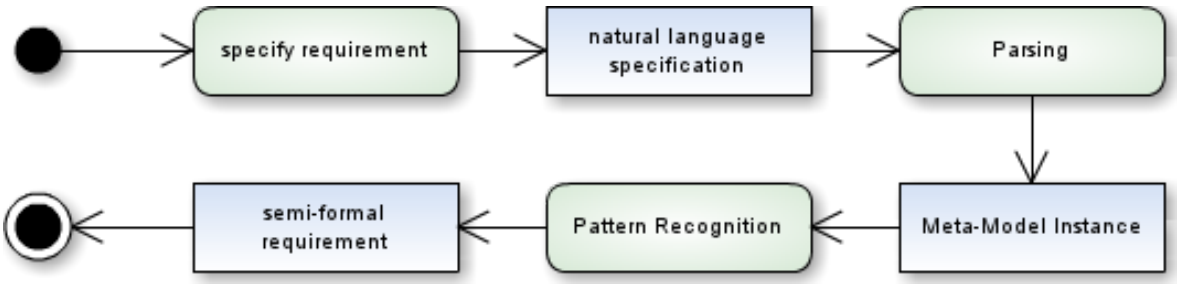

Figure 7.1: Conceptual workflow of the prototype.

7.2.1 Parsing

In linguistics, parsing describes the decomposition of sentences grammatical structure in order to derive a respresentation of their semantic and syntactic structure [Sla13]. A parse tree can be used to visualize the decomposition precedure step-by-step, as depicted in chapter 4. In computer science, a lexical analyzer, which is oftened shortened as lexer, preprocesses the sentence for the parser [ALSU86]. The interaction between a lexer and a parser is visualized in Figure 7.2. The lexer is responsible for two tasks:

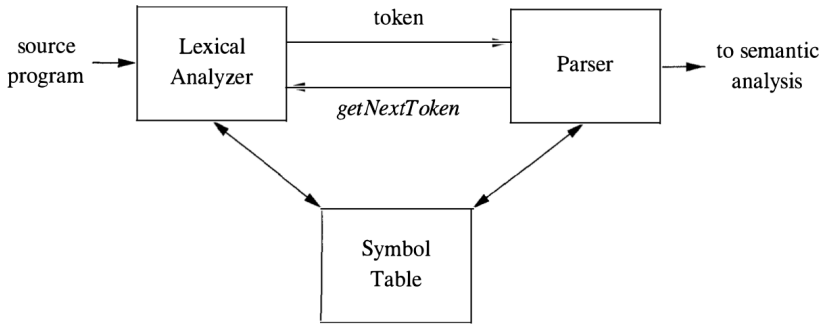

Figure 7.2: Token-based interactions between Lexer and Parser. [ALSU86]

- *Scanning* – The lexer scans the input as plaintext and removes previously defined parts of the input, such as comments in sourcecode, multiple whitespaces or other semantically unimportant parts.
- *Lexical Analysis* – The lexer analyzes the textual input, such as sourcecode or natural language specifications and replaces the textual charactersets with tokens, which are defined in the symbol table. The symbol table includes keywords, definitions of charactersets which define what a *word* or a *number* is in the context of the lexer and parser. The symbol table can also contains charactersets with constraints, such as variable names in java, which consists of alphanumerical characters, but must start with a letter [Ora13].

A parser is written for a specific language, which is usually defined by a context-free grammar or a regular expression. The parser iterates over the tokens offered by the lexer and evaluates, if the chain of tokens can be created using a given grammar. If it can be created, the input is successfully parsed. The parser can define semantic actions, which are executed while parsing and can directly interpret the language or make a statement about the input's structure. Within the prototype, semantic actions transfer the natural language specification into a datastructure based on the meta-model in section 6.

7.2.1.1 JavaCC

The prototype uses the Java Compiler Compiler (furthermore JavaCC) in order to parse the input. JavaCC is a parser generator and lexical analyzer which generates java classes, which can parse the given language [Nor13]. It was chosen due to its user friendliness and its good documentation. In addition, an employee at the university of oldenburg was able to give support when Parsers generated by JavaCC cannot handle left recursion, such as $A := A, B$ in its grammar rules. Parsers, which are generated by JavaCC, throw a ParseException, if the input is not part of the parsers language. Otherwise the parsing operation runs successfully without further feedback. The language is defined in a single file which contains the Java class definition, the token definitions for the lexer, and the parsing rules [Jav13].

The structure of a JavaCC file is shown in Table 7.1. JavaCC offers many *options* which can be set to true or false, in order to enable complex features such as detailed debug output for the parser and the lexer, or unicode support. *SKIP-Tokens* define, which charactersets are removed within the *scanning* task of the lexer. The content of the symbol table for the parser and lexer are directly defined as *tokens*. The *parse() method* is the entry point for the language definition as regular language.

Section	Explanation
Options	Configuration flags for JavaCC, such as debug output.
SKIP-Token	Defines, which characters are to be ignored by the parser.
Tokens	Defines, which character sequences are represented as tokens by the lexer.
parse() Method	The parsing entry point. contains regular expressions and java code.

Table 7.1: Structure of a JavaCC file

7.2.1.2 JavaCC Lexer

Each token defined JavaCC's Lexer is an object and has two fields: *type* holds the tokens name; *image* contains the represented text. Figure 7.3 represents an example lexer in JavaCC with token definitions. **SKIP** in line 2 is a special token, which defines the symbols removed from the input in the *scanning* task. The **SKIP**-Token's content is removed after the lexical analysis and before the input is offered to the parser. Thus characters within the **SKIP**-Token may be used within other tokens. Ordinary *TOKENs* in lines 4-10 are split into keywords, such as **if**, **then** or **continuously** and generic definitions for **letters**, **words** and **numbers**. **EOL** represents the "End of Line" character in all JavaCC regular expressions.

Lexer example in JavaCC

```
1  // TOKEN : { < $tokenname$ : $regex$ > }
2  SKIP : { " " | "\t" | "," | "." }
3  // Keywords
4  TOKEN : { < IF : "If" | "When" > }
5  TOKEN : { < THEN : "then" | ", then" | "," > }
6  TOKEN : { < DURING : "during the next" > }
7  // Generic definitions
8  TOKEN : { < WORD : (<LETTER>)+ > }
9  TOKEN : { < LETTER : (["A"-"Z"] | ["a"-"z"])+ > }
10 TOKEN : { < NUMBER : (["0"-"9"])+ > }
```

Figure 7.3: Lexer example in JavaCC

7.2.1.3 JavaCC Parser

An excerpt of a JavaCC parser is depicted in Figure 7.4. Lines 1 to 25 depict parsing rules, which contain regular expressions. The *parse()* method in line 1 is the starting point. Tokens in regular expressions are embedded in angle brackets, like <*IF*> in line 7.

LOOKAHEAD(50) in line 9 applies to the *or* between _dualTrigger() and _simpleTrigger() and specifies, that the parser simulates 50 tokens into the future before determining, which of the rules applies. In this context, it means that a _simpleTrigger() can be 50 words in size. Because _dual-Trigger() begins with a _simpleTrigger(), the parser cannot tell the difference without looking ahead until it recognizes, if an <*AND*> token exists. The precise behaviour of LOOKAHEAD() in JavaCC is explained in the JavaCC documentation [Nor13].

Semantic actions in JavaCC are Java-Blocks, which are embedded in curly brackets, as seen in lines 29, 33 and 35. The method *getToken(int)* gets a token from the Lexer within a semantic action. GetToken(0) returns the last token, which was read by the parser. By using a number greater than 0 as parameter, tokens can be retrieved from the lexer, before they are read by the parser itself. This must be used with caution, as these tokens are not parsed yet and the tokens can be used within semantic actions, even though they might result in a parsing error, when the parser reads them.

Parser example in JavaCC

```
1  void parse() : {}
2  {
3          _triggerImplies()
4  }

5  void _triggerImplies() : {}
6  {
7          <IF>
8          (
9                  LOOKAHEAD(50)
10                 _dualTrigger()
11           |
12                 _simpleTrigger()
13         )
14         ( <DURING> )?
15 }

16 void _dualTrigger() : {}
17 {
18         _simpleTrigger()
19         <AND>
20         _simpleTrigger()
21 }

22 void _simpleTrigger() : {}
23 {
24   _predicate()
25 }

26 // output all tokens in the predicate as String
27 void _predicate() : {}
28 {
29         { String temp = new String(); }
30         (
31           LOOKAHEAD(10)
32                 ( <WORD> | <OR> )
33                 { temp = temp + " " + getToken(0); }
34         )+
35         {         System.out.println(temp.trim()); }
36 }
```

Figure 7.4: Parser example in JavaCC

7.2.2 Pattern Recognition

Using the parser from section 7.2.1 and the meta-model defined in section 6, natural languages specifications can be analyzed for their pattern conformity. The complete workflow is depicted in Figure 7.5. This workflow is demonstrated by example of the natural language specification *If it rains for 1 minute, then the wipers are activated within 30 seconds until the windscreen is dry.*.

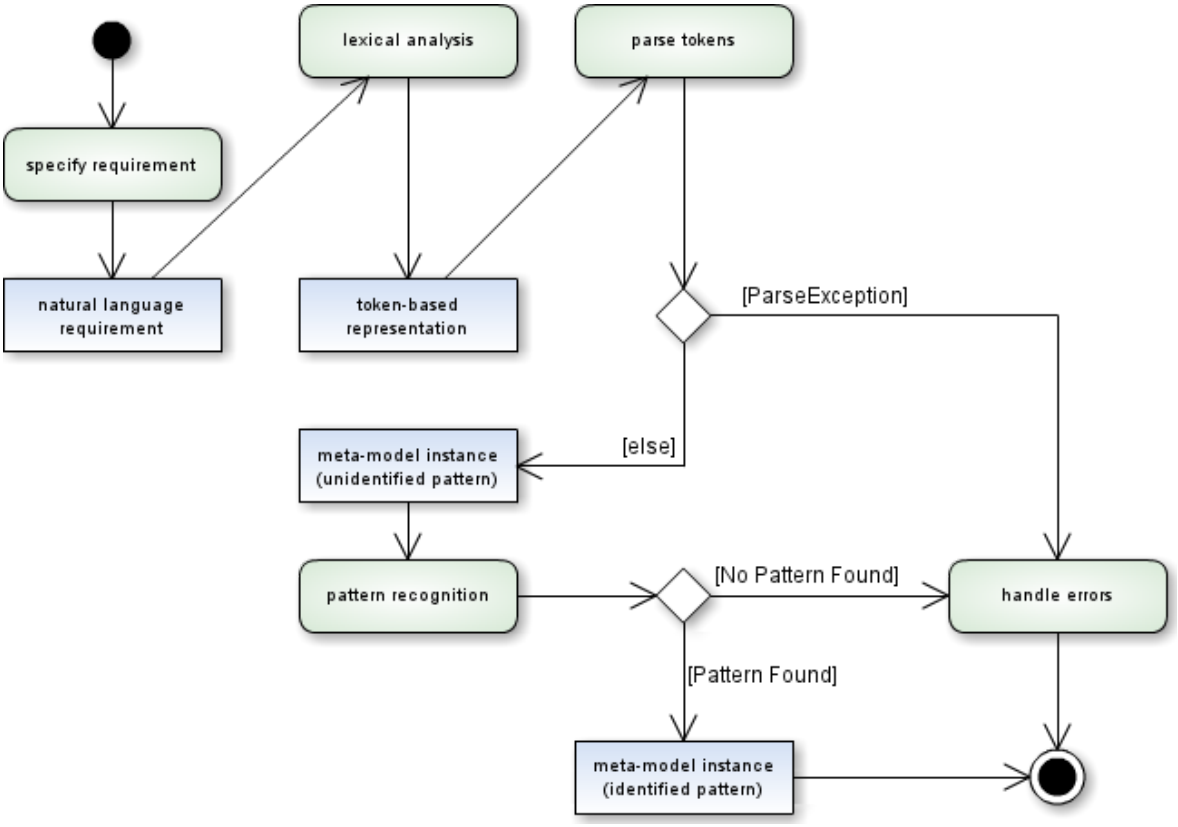

Figure 7.5: Pattern recognition workflow as UML activity diagram.

Initially, the lexer receives the natural language specification as input. The example contains a punctuation, which is removed by the scanning process, leaving *If it rains for 1 minute then the wipers are activated within 30 seconds until the windscreen is dry* to be transfered to tokens. The keyword *if* is represented by the token <*IF*>, *for* is represented by <*DURING*>, *seconds* and *minute* are represented by <*TIME_UNIT*>, *then* is represented by <*THEN*> and *until* is represented by <*UNLESS*>. Every other word or number is represented by a respective generic token <*WORD*> or <*NUMBER*>. The token-based representation offered by the lexical analysis is depicted in Figure 7.6. The tokens values, which are the corresponding substrings from the natural language specification, can be accessed via the token's *image* field, which was mentioned in section 7.2.1.2.

<*IF*> <*WORD*> <*WORD*>
<*DURING*> <*NUMBER*> <*TIME_UNIT*>
<*THEN*> <*WORD*> <*WORD*> <*WORD*> <*WORD*>
<*FINALLY*> <*NUMBER*> <*TIME_UNIT*>
<*UNLESS*> <*WORD*> <*WORD*> <*WORD*> <*WORD*>

Figure 7.6: Result of the lexical analysis.

The first action carried out by the parser is the identification of the activation mode and start-phase. As this example contains neither, the default activation mode *CYCLIC* and the default start-up phase *IMMEDIATE* are chosen. Next the parser recognizes the kernel patterns base structure which, in this case, is <*Trigger*> <*Action*> <*Time*>. The parser then aggregates the <WORD>

tokens to <PREDICATE> tokens and <NUMBER> <TIME_UNIT> tokens to <INTERVAL> tokens. The intermediate result is shown in Figure 7.7.

Activation Mode: CYCLIC
Start-Up Phase: IMMEDIATE
Trigger: <*IF*> <*PREDICATE*> <DURING> <INTERVAL> <THEN>
Action: <PRECIATE>
Time: <FINALLY> <INTERVAL> <UNLESS> <PREDICATE>

Figure 7.7: Intermediate parsing result.

The parser can now identify the structure of the *trigger*, *action* and *time* components of the kernel pattern in order to create an instance of the meta-model. The tokens <IF> and <THEN> are aggregated to the <IF_THEN> token. All other tokens are representable with the meta-model. The unidentified kernel pattern component of the instance is depicted in Figure 7.8.

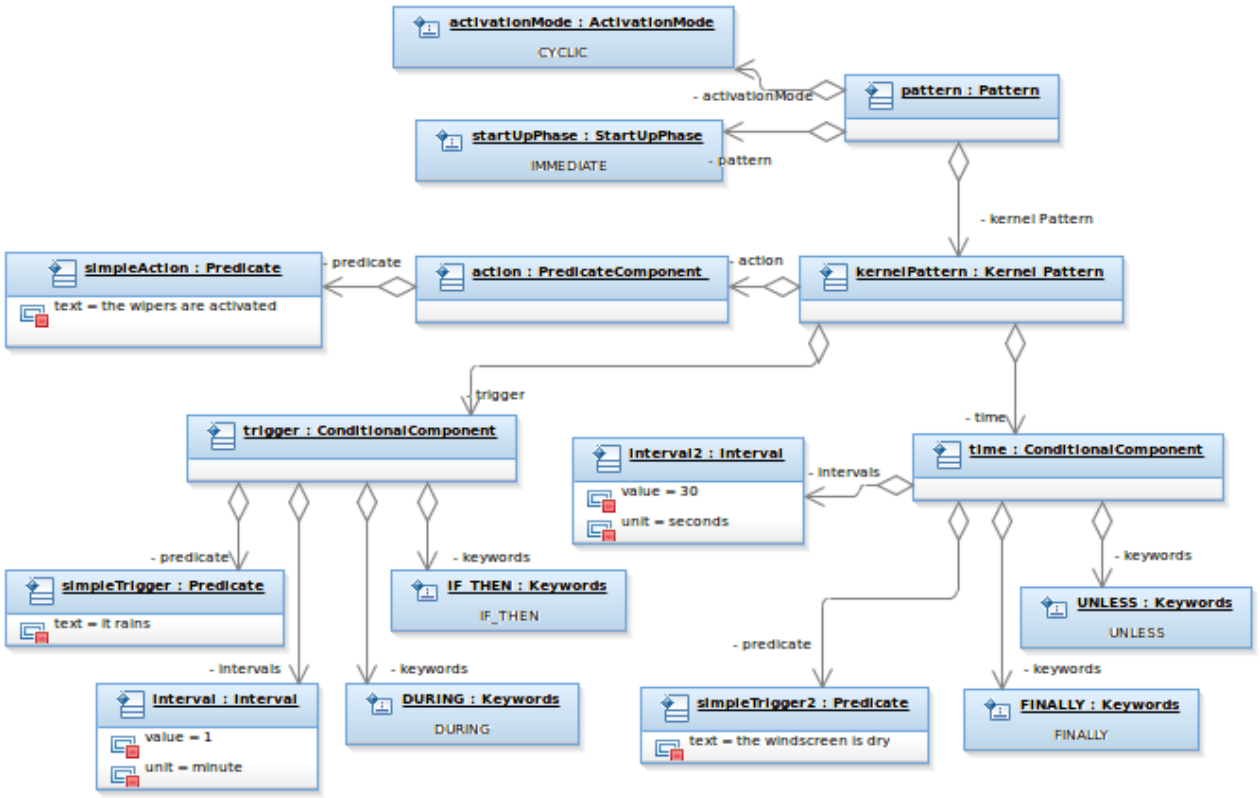

Figure 7.8: Meta-model instance generated by the parser.

The Prototype now compares the meta-model instance with the structure of each kernel pattern defined by BTC EmbeddedSystems AG. This is done by comparing the core properties, which are the quantity of predicates and intervals, aswell as the occuring keywords. If a kernel pattern has exactly the same amount of components, a match is detected and the predicates are mapped to the kernel patterns macros based on their location in the natural language specification. The result is the pattern

including activation mode and start-up phase aswell as the mapped macros. This information, which is visible in Figure 7.9, can be entered into the BTC EmbeddedSpecifier.

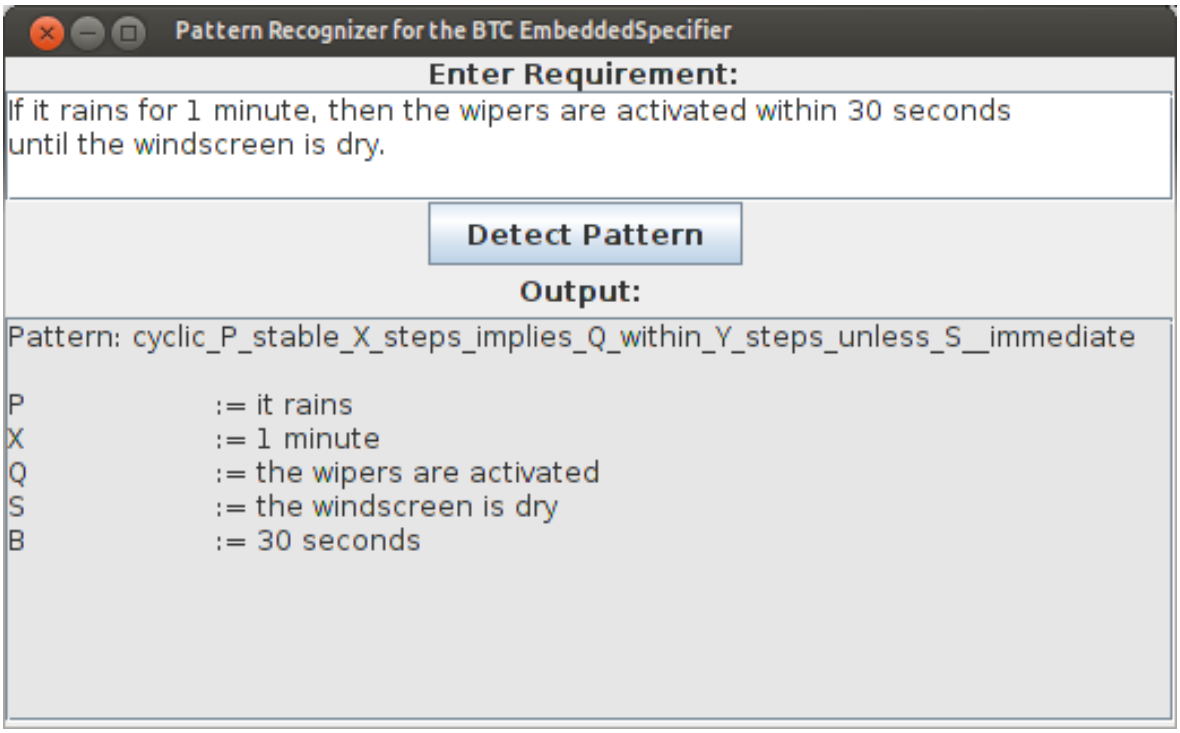

Figure 7.9: Final result of the pattern recognition process.

7.3 Implementation

The prototype implements the workflow explained in section 7.2.2. Section 7.3.1 explains how the prototype was developed. Section 7.3.2 describes the prototype's architecture, whereas section 7.3.3 describes the prototype from a user perspective.

7.3.1 Development Process

As the prototype uses JavaCC, the parser's code is autogenerated. This autogenerated code is hard to read or to verify. Thus it was considered a blackbox component during development. A test-driven approach was used throughout the development process, in order to identify errors with the parser. A unit test was written for every linguistic structure, which was to be accepted by the parser. Any faulty behaviour by the parser could be identified by the results of the several test cases and could be isolated efficiently. This property is vital when extending the parsers grammar, as debugging a grammar without test coverage is a very difficult task. The entire prototype was developed strictly using test-driven development to ensure a high test coverage. The workflow of test-driven development is depicted in Figure 7.10 and consists of three steps: *red*, *green* and *refactor*.

The first step is to write a test, which fails. This is known as a *red* test. Then the code is written to conform to make this test case succeed. A successful test is known as a *green* test. After the test

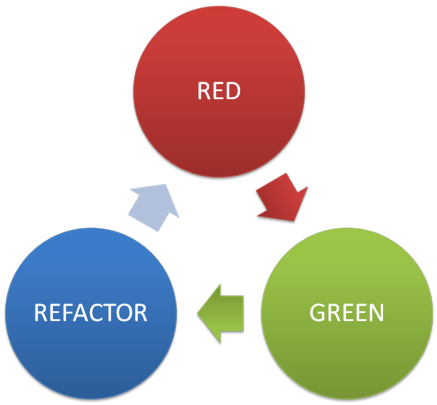

Figure 7.10: Test-driven development workflow. [Hey13]

succeeded, the code is refactored in order to become extendable and maintainable. These three steps are repeated, until the code has implemented the required functionality. This development cycle was applied for all kernel patterns and for every linguistic structure, which the prototype supports.

7.3.2 Architecture

This section outlines the prototypes architecture. This includes its classes along with their public interface, as well as relations to other classes nd is visualized in the UML class diagram in Figure 7.11. This class diagram omits irrelevant details and displays additional details, which increase understandability.

The entry point is the *main()* method within the *JusticePrototype* class. From here, the Window is created as a *JFrame*, which contains various graphical elements, most of which are unimportant for the prototypes functionality. Among these graphical elements, is the *Detect Pattern*-Button, which initiates the workflow on press.

The *DetectButtonActionListener* triggers the pattern recognition. This is done in two processes, which are encapsulated in two classes: *PatternParser* and *PatternMatcher*.

The *PatternParser* contains the automatically generated code by JavaCC, thus offering the parse() method.As JavaCC's *parse()* method must return void, a second method, *getModel()* returns the metamodel compliant model as output of the parsing process. *PatternParser* contains further automatically generated methods, which are publicly accessible. Due to their large quantity and their low expressiveness, they are omitted from this class diagram.

The *PatternMatcher* holds a structural representation of every *KernelPattern* within the BTC EmbeddedValidator Pattern Library Release 3.7 in a list. This list is created by the *KernelPatternStructureFactory*, which holds the structure of all *KernelPatterns*.The *match()* method compares the model from the PatternParser with every KernelPattern in the Pattern Library to find structural equality.

The *KernelPatternStructureFactory* creates empty instances of *KernelPatterns*, which solely represent the structure of these. This factory class contains the structural information of all *KernelPatterns*. In order to extend the prototype by more *KernelPatterns*, their structure must be defined within this factory. A thorough explanation on extending the prototype is given in section 8.

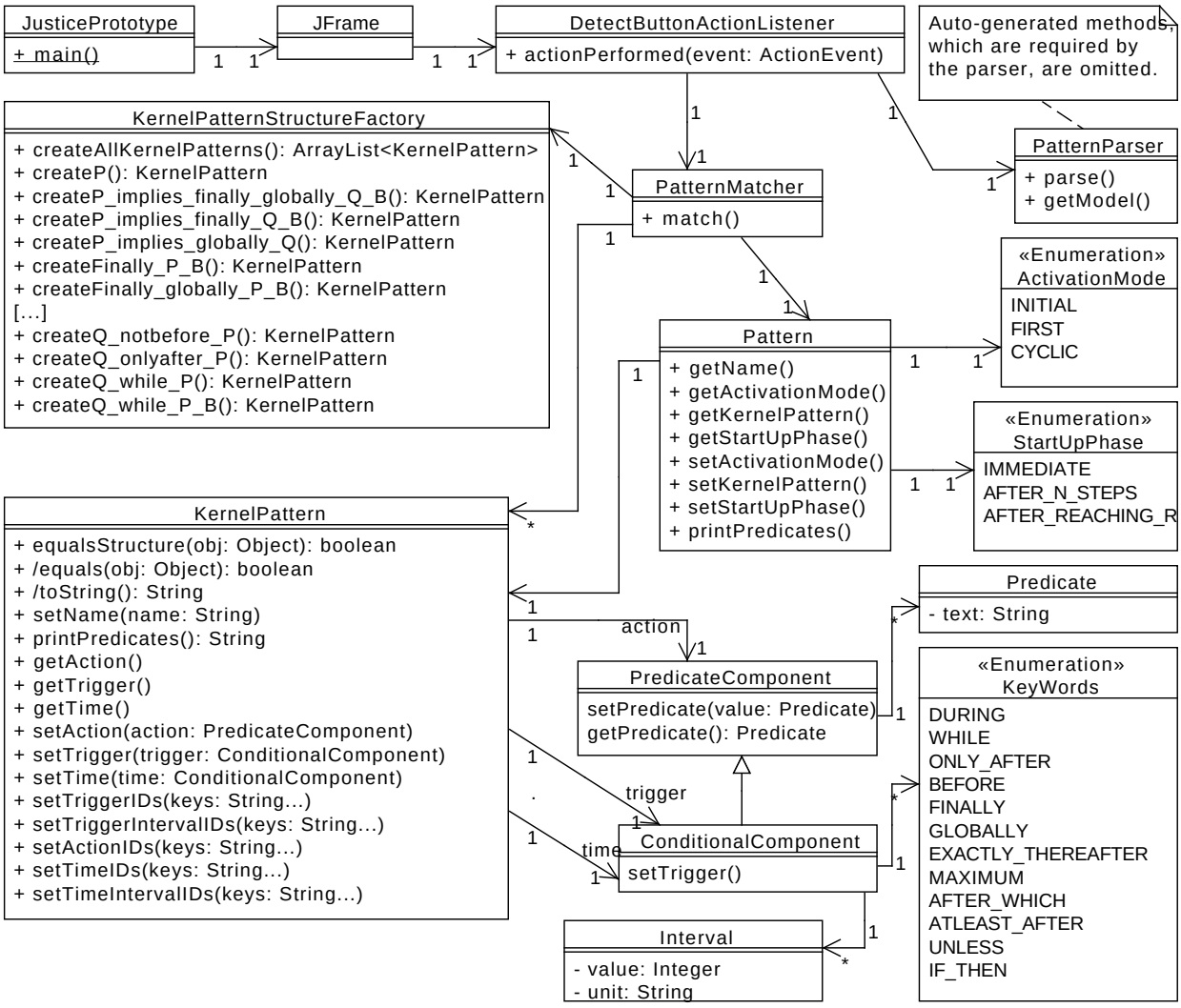

Figure 7.11: UML class diagram of the prototype.

Every *Pattern* consists of an *ActivationMode*, a *StartUpPhase*, and a *KernelPattern*, as described in section 6. A *Pattern* instance does not have a name after its creations, but instead receives a name after it has successfully been matched with a *KernelPattern*.

The available *ActivationMode*s and *StartUpPhase*s are implemented as enumerations.

The *KernelPattern* offers many methods to set and change its *Trigger*, *Action* and *Time* Components. Every component within the *KernelPattern* has an ID, which represents the order in which the *PatternParser* returned the components. These IDs are used to map the components to the Identifiers defined in the Pattern Library, such as P, Q, S and X.In addition to these, it contains an overwritten *toString()* method to offer formatted output. Every *KernelPattern* can be compared with other *KernelPattern*s in order to determine structural equality or exact equality with the *equalsStructure()* and *equals()* methods respectively.

*PredicateComponent*s are containers for an arbitrary number of *Predicate*s, which contain the text, which the *Predicate* represents. A *KernelPattern*'s *Action* component is a *PredicateComponent*.

*ConditionalComponent*s are containers which can hold an arbitrary number of *KeyWords* and *Intervals*. *ConditionalComponent*s extend *PredicateComponent*s and thus also contain an arbitrary number of *Predicate*s. A *KernelPattern*'s *Trigger* and *Time* components are *ConditionalComponent*s.

7.3.3 Final Prototype

This section describes the prototype from the user's point of view. Figure 7.12 depicts the prototypes user interface (UI), as it is presented to the user after startup. The UI is kept very simple and consists of three components:
The *Input Area* accepts the users input. The input can be split into multiple lines to maintain clarity in larger natural language specifications. Pressing the *Detect Pattern*-Button inititiates the parsing of the string in he *Input Area*. The output of the parsing process is written to the *Output Area*.

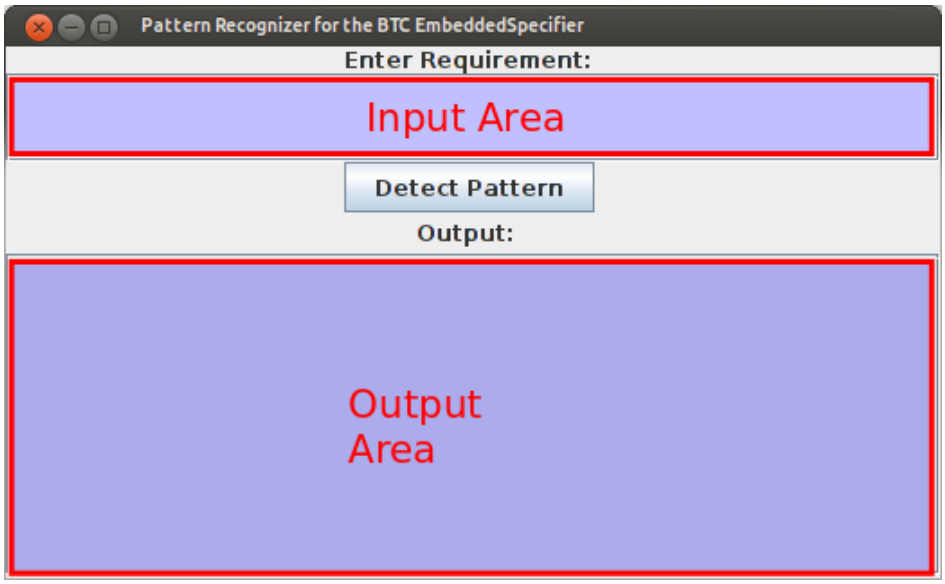

Figure 7.12: Graphical user interface of the prototype.

Figure 7.13 depicts the UI after an input text was entered and the *Detect Pattern*-Button was pressed: The *Output Area* displays the identified Pattern including the activation mode and start-up phase, aswell as the Macros and their corresponding text from the natural language specification.

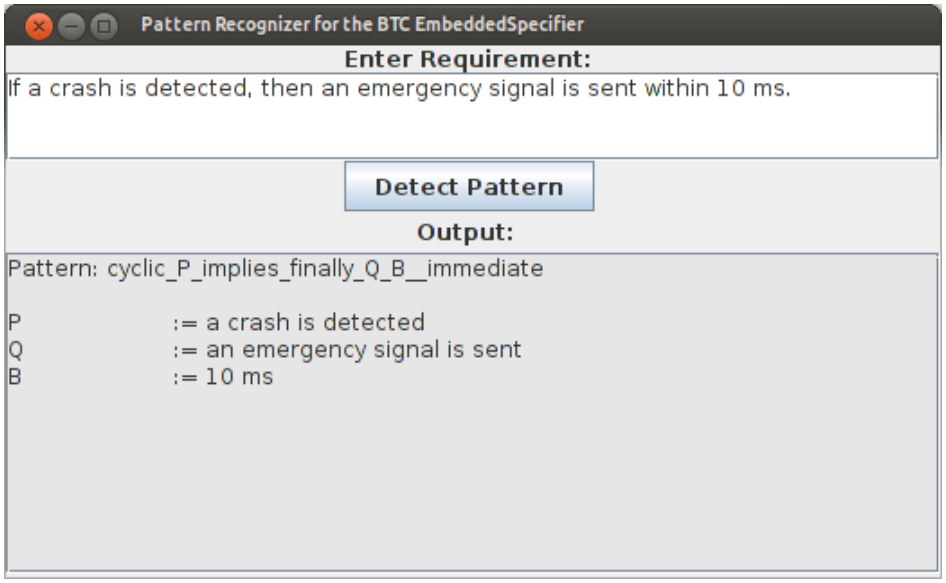

Figure 7.13: Input and output of the prototype.

8 Extending the Prototype

This section describes the extension of the prototype by a new kernel pattern. As the prototype was developed using a test-driven approach, the extension of the prototype is test-driven aswell. The prototype can be extended at two points: the grammar and the meta-model. Figure 8.1 gives an overview of the workflow required to extend the prototype. This workflow can be applied wether the intent is to extend the prototype by a kernel pattern, or to extend the grammar by synonyms or linguistic structures. The extension of the grammar can be done using this approach, as the test will subsequently correctly identify a pattern and thus complete successfully.

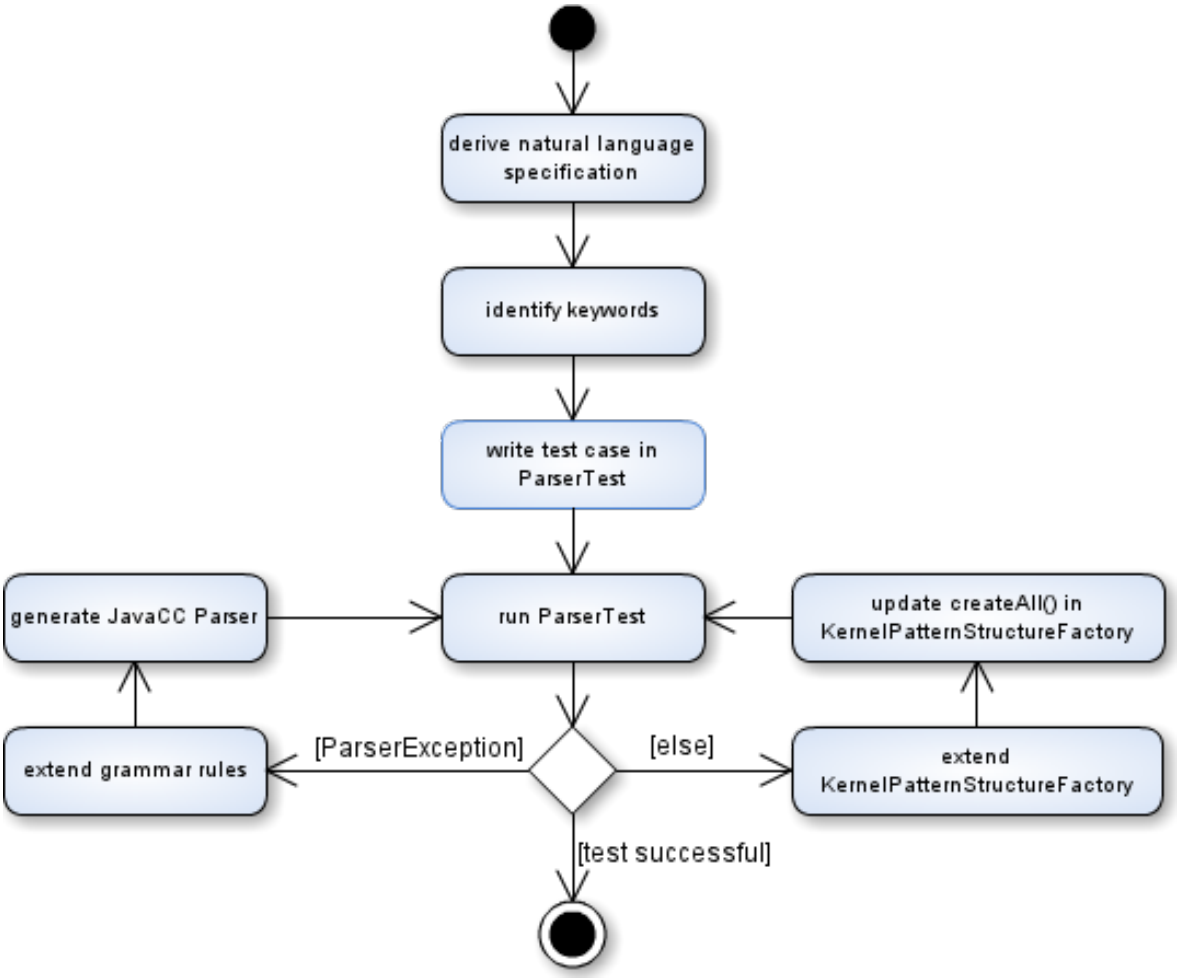

Figure 8.1: Test-driven workflow for extending the prototype by a kernel pattern.

Firstly the linguistic structure of the natural language specification for the new kernel pattern must be identified. This can be done by writing natural language requirements, whereby their logic must comply to the new kernel pattern's büchi automaton. Among these natural language requirements, common structures must be identified. This process was applied in chapter 4.

After a common structure has been found, keywords for tokens within the parser must be identified. New keywords should only be defined then, when existing keywords cannot be reapplied to the new context.

As the prototype was developed with a test-driven approach, the extension by a new kernel pattern occurs test-driven aswell. The developer must *write a test case for every common structure in ParserTest*. Each test case simply calls the parse(String) method with the common structure as parameter. Subsequently the parsed model's kernel pattern's name is compared to the new kernel patterns name. The existing test cases can be used as a reference for this procedure.

The further procedure is determined by the result of the next test run. The test can succeed, fail due to a falsely identified kernel pattern or fail due to a ParserException. In the case of a ParserException, the grammar must be extended as explained in section 8.1. In the case of a falsely determined kernel pattern, the KernelPatternStructureFactory must be extended as explained in section 8.2. This procedure is repeated until the test run is successful. In the case of a succesful test run, the new kernel pattern is fully implemented.

8.1 Extending the Grammar Rules

If the ParserTest fails with a ParserException, the input's structure is not covered by the grammar. The grammar is split into three main components: *trigger*, *action* and *time*. These are explained in chapter 5 and are easily extendable, as they constitute of *OR*-constructions.

After extending the grammar rules, the JavaCC Parser's code must be synchronized to accept the same input. As the parser's code is in a different, yet functionally equivalent format, adopting the new grammar rules is straight-forward. The parser must be furthermore be extended by semantic actions, which *build* the meta-model instance. The parser is generated by running the commandline operation *javacc *.jj* followed by compiling the complete prototype including the generated .java files.

8.2 Extending the KernelPatternStructureFactory

If the ParserTest fails without a ParserException, the parsed kernel pattern's name doesn't correspond to the expected name. This indicates, that the kernel pattern is not properly defined in the *KernelPatternStructureFactory*. The *KernelPatternStructureFactory* contains a factory method for each kernel pattern with the naming convention *create<KernelPatternName>()*. Every factory methods create an empty instance of a *KernelPattern* with a structurally correct *Trigger*, *Action* and *Time*. The *PredicateComponent* and *ConditionalComponent* classes have constructors to create *empty instances*. See existing factory methods within the KernelPatternStructureFactory as reference.

After adding the factory method in the *KernelPatternStructureFactory*, the factory method must be used in order to add an empty instance of the kernel pattern to the ArrayList, which is built within the *createAll()* method. If the kernel pattern's structure was correctly defined in the factory method, the prototype will now correctly identify the parsed model's kernel pattern structure and apply the name to it, thus resulting in a succesful test run.

9 Outcome

This chapter evaluates, if the prototype created in chapter 7 solves the problem defined in chapter 1.2. Section 9.1 validates the prototype's efficiency with fictional, as well as real-world examples as input. Section 9.2 adresses the freedom of natural language and the resulting problem of interpretation. Section 9.3 gives a brief summary of what has been achieved in the thesis as well as the limits of the approach. Section 9.4 gives an overview of possible fields for further research.

9.1 Validating the Prototype

This section validates the prototype's efficiency by testing the fictional requirements from chapter 4 as input in section 9.1.1. Subsequently real-world example requirements from customers of BTC EmbeddedSystems are tested in section 9.1.2.

9.1.1 Fictional Examples

The prototype was based on the grammar defined in chapter 5. This grammar was mostly based on the example requirements defined in chapter 4. In order to validate the prototype, it is necessary to evaluate, if the prototype can correctly identify the example requirements, which were the origin of the grammar, to determine if the prototype eached its goal. The example requirements were entered into the input mask of the prototype. The prototype proofed to identify all example requirements and map their macros correctly. The prototype had problems when encountered by a newline or hyphenation, which had to be removed. Newlines were replaced with whitespaces before the input was passed on to the parser, in order to resolve the issue with newlines. Hyphens remain a problem, because they are followed by a newline, which splits the word. These must be resolved by hand. Apart from these two issues, the prototype successfully recognized the patterns including the corresponding macros for all example requirements defined in chapter 4.

9.1.2 Real-World Examples

As the BTC EmbeddedTester is a product for customers, the formalization approach within this thesis should ideally process real requirements formulated by customers as well. This approach was tested with input data from three documents from customers of BTC EmbeddedSystems. For reasons of confidentiality, the documents and their content is not described. The content of requirement excerts within this section are purely fictional, whereas the original requirement's linguistic structure was preserved.

The first problem were symbols and numbers within predicates in the requirements. These were vital, as they were used to refer to other requirements with identifiers such as <*REQ_5*>. The parser did not accept words with symbols and numbers, as the fictional examples only contained letters. As the approach uses a grammar, it was easily extended to accept words with the following rule: *WORD = LETTER (LETTER*> | <*NUMBER)**.Thus a word within the parser can begin with a letter followed by an arbitrary amount of letters or numbers.

The second major problem was the occurence of keywords within the natural language specifications. In some cases, it was sufficient to replace the keyword-occurence with a synonym. In some cases, however, it was impossible to remove the keyword-occurence without altering the requirement's linguistic structure. The most problematic keywords are *while* and *and*, as these can easily occur within a *trigger* in natural language specifications. A workaround would be to summarize two signals into one, so that the parser doesn't attempt to parse a *DUAL_TRIGGER* rule.

Common linguistic structures within the requirement documents are short implications, which are very unambiguous and easy to parse. These requirements are the best-case scenario for this approach and lead to a very easy formalization. A possible drawback might be, that these very short requirements require a large amount of requirements to offer complete requirement coverage.

The most common structure within one document was the *cyclic_P_triggers_Q_unless_S__immediate* pattern. This pattern represents an implication with a temporal constraint in form of a signal *S*. The requirements within this document all had similar structures, which suggests that these requirements could have been created using boilerplates or similar structural constraints. As such, many requirements could be parsed efficiently. About one third of the requirements had the temporal constraint formulated in a structure, which was not recognized by the parser. The grammar can be altered to accept this linguistic structure. It was not implemented in the prototype of this thesis, as the rules had a conflict with existing rules to represent temporal constraints. Thus, this new structure would *replace* the old structure.

Many requirements described components and their basic functionality, as well as their communication with other components. These requirements were mostly formulated in the structure of a pattern cyclic_P__immediate. An example for such a requirement would be *Component A sends its output to Component B via TCP/IP*. These requirements are very high level and do not contain signals, which can be mapped to a signal within a target architecture. In order to efficiently formalize these requirements, many more details, especially those about the used signals for the implementation, are required. As the input and output signals must be mapped within a formal requirement, a clear interface definition is sufficient.

9.2 Freedom of natural language specifications

Natural language offers freedom in its formulation and its interpretation. This makes it very difficult to automatically parse and interpret natural language specifications. Even an optimal parser, which could automatically recognize and interpret all natural language specifications would be challenged by the ambiguity of natural language. While the ambiguity can be reduced by identifying the context of the natural language specification, it cannot be eradicated. The prototype in section 7 only processes a very limited spectrum of natural language, which is limited by the grammar. Due to this restriction in functionality, ambiguity is minimized for the limited spectrum, thus enabling a precise mapping to a semi-formal requirement and its macros.

9.3 Conclusion

The goal of this thesis was to automate the transition from natural language specifications to semi-formal requirements in context of the BTC EmbeddedSpecifier. The prototype created in section 7

successfully parsed the fictional example requirements and a significant amount of real-world example requirements into semi-formal requirements, thus fulfilling the required basic functionality. As stated in section 9.2, ambiguity poses a great challenge for automatic interpretation.

The prototype only accepts a strongly limited set of natural language as its input, in order to minimize ambiguity without interpreting the input. This allows for the prototype to precisely identify patterns and macros, but it also means that the amount of accepted natural language specifications is very limited. As a result, many natural language specifications within the real-world example requirements were not parsed, although some of them were very unambiguous. When considering the complete set of natural language specifications possible, it is also very difficult to precisely identify a single applicable pattern. Many natural language specifications can be mapped to several patterns, before considering ambiguity.

As the prototype is based on a grammar and a meta-model, its limited input spectrum can be extended according to the customer's needs. By extending the grammar to support the structure of the customer's natural language specifications, the boundaries of the input limitations can be shifted towards the customer's needs. Not only can the boundaries be shifted, but they can also be broadened to accept a much wider spectrum of natural language specifications than the current prototype. Also th meta-model is not bound to its current set of kernel patterns. Kernel patterns can be added or changed as described in section 8. This can also be used to support an alternative to the current kernel pattern system, by defining a new meta-model. In conclusion, the approach can be adopted to the need of each individual user and the input spectrum can be broadened as required.

9.4 Further Research

The outcome of this thesis reveals the following topics for further research: The grammar contained in this thesis is not optimal for all customers. Although customers have specific needs and very different linguistic structures within their natural language specifications, an optimal generic grammar could be constructed in order to fulfill most needs. However this would require a large amount of customer's requirements. Apart from the problematics of natural language specifications in english or other western languages, the grammar does not take asian languages and their linguistic structures into account. Japanese, for example, has a very complex linguistic structure when compared to european languages. Thus it is not evaluated if this approach is efficient with asian languages. The approach can also be extended to interpret the natural language specification in order to directly identify ambiguity and react accordingly. The approach could prioritize applicable patterns according to different factors. Another possibility of research is the replacement of the meta-model. BTC EmbeddedSystems has a generic alternative to the kernel pattern library, which was used within this thesis. This thesis does not adress the replacemenet of the kernel pattern meta-model with an alternative meta-model. While the approach is metamodel-generic, it would have to be adopted to support such an alternative meta-model.

9.5 Related Work

In his master's thesis "CESAR - text vs. boilerplates: What is more effcient - requirements written as free text or using boilerplates (templates)?" Vegard Johannessen analyzed the differences between free text requirements and boilerplate requirements [Joh12]. His approach reduces ambiguity in natural

language specifications by using boilerplates, as opposed to the grammar-based approach within this thesis, which resembles limited free text. In his thesis, Johannessen did not come to an obvious conclusion, whether boiler plates or free text requirements are to be preferred. He did note that boiler plates have the advantage of producing requirements with less complexity. Due to the reduction of the spectrum of natural language specifications, the complexity is reduced with the approach introduced in this thesis as well. As such, the grammar-based approach does not suffer from the disadvantages of free text requirements and could be compared to boilerplate requirements in terms of complexity. In order to determine, if a grammar-based or a boilerplate approach have any advantage over the other further research would be necessary.

10 Appendix

This chapter contains figures, which are omitted to not impede the readability of the thesis.

Final Grammar

Extended grammar used in the prototype

```
PATTERN = [ACTIVATION_MODE], KERNEL_PATTERN, [START-UP];

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Activation Mode
ACTIVATION_MODE = INITIAL_AM
                | FIRST_AM;

INITIAL_AM = "initially" | "Directly after start-up" | "After start-up";
FIRST_AM = "For one occurence" | "For the first occurence" | "For the first time";

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Start-Up Phase
START-UP  = AFTER_REACHING_R
          | AFTER_N_STEPS;

AFTER_REACHING_R = "After", INTERVAL;
AFTER_N_STEPS = "After", PREDICATE;

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Kernel Pattern
KERNEL_PATTERN =
    TRIGGER, ACTION, [TIME]
    | ACTION, [POST_TRIGGER]
    | ACTION, TIME, [POST_TRIGGER]
    | [TIME], ACTION, TRIGGER;

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Trigger
TRIGGER =
    IF, TRIGGER_IMPLIES, [THEN]
    | TRIGGER_IMPLIES, IMPLIES;
POST_TRIGGER =
    IF, TRIGGER_IMPLIES;
TRIGGER_IMPLIES =
    (DUAL_TRIGGER | SIMPLE_TRIGGER), [DURING];
DUAL_TRIGGER =
    SIMPLE_TRIGGER, AND, SIMPLE_TRIGGER;
SIMPLE_TRIGGER =
    PREDICATE;
STABLE =
    IF, SIMPLE_TRIGGER, DURING, THEN;
PREDICATE =
    (WORD)+;
IF =
    "if" | "when";
THEN =
    "then" | ", then" | ",";
IMPLIES =
    "implies that" | "imply that";



// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Action
ACTION =
    SIMPLE_ACTION;
SIMPLE_ACTION =
    PREDICATE;
```

```
45  // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - Time
46  TIME =
47      ( ONLY_AFTER | NOT_BEFORE )
48      |
49      ( FINALLY, AFTER_WHICH, FINALLY )
50      |
51      ( (GLOBALLY, [FINALLY]) | (FINALLY, [UNLESS]) )
52      |
53      ( ATLEAST_AFTER )
54      |
55      ( [DURING], EXACTLY_THEREAFTER )
56      |
57      ( DURING, [FINALLY | AFTER_WHICH_OR] )
58      |
59      ( UNLESS, [FINALLY] )
60      |
61      ( [MAXIMUM_INTERVAL], WHILE );

62  WHILE =
63      WHILE_WORD, SIMPLE_TRIGGER;
64  WHILE_WORD =
65      "while" | "as long as" ;
66  MAXIMUM_INTERVAL =
67      MAXIMUM, INTERVAL;
68  GLOBALLY =
69      "continuously";
70  FINALLY =
71      WITHIN, INTERVAL;
72  ATLEAST_AFTER =
73      ATLEAST, INTERVAL, AFTER, DUAL_TRIGGER;
74  DURING =
75      DURING_WORD, INTERVAL;
76  EXACTLY_THEREAFTER =
77      EXACTLY, INTERVAL, THEREAFTER;
78  UNLESS =
79      UNLESS_WORD, (SIMPLE_TRIGGER | DUAL_TRIGGER);
80  AFTER_WHICH_OR =
81      AFTER_WHICH, MAXIMUM_INTERVAL, OR, UNLESS;
82  AFTER_WHICH =
83      AFTER_WHICH_WORD, SIMPLE_TRIGGER;
84  ONLY_AFTER =
85      "only after", SIMPLE_TRIGGER;
86  NOT_BEFORE =
87      "before", SIMPLE_TRIGGER;



88  INTERVAL =
89      NUMBER, TIME_UNIT;
90  NUMBER =
91      {DIGIT}-;
92  DIGIT =
93      "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
94  WORD =
95      LETTER {LETTER | NUMBER};
96  TIME_UNIT =
97      (("nanosecond" | "millisecond" | "second" |
98      "minute" | "hour" | "day" | "step"), ["s"])
99      | s | ms | ns | m ;
100 WITHIN =
101     "within" | "after";
102 MAXIMUM =
103     "for a maximum of";
104 ATLEAST =
105     "at least";
106 AFTER =
```

```
    "after";
AND =
    "and";
OR =
    "or";
DURING_WORD =
    "for" | "for the next" | "during the next";
EXACTLY =
    "exactly" | "precisely";
THEREAFTER =
    "thereafter" | "later";
UNLESS_WORD =
    "until";
AFTER_WHICH_WORD =
    "after which";
```

Meta-Model Compliant Models for Kernel Patterns

This section contains an instance of the meta-model in section 6 for each of the 24 kernel patterns defined in the BTC EmbeddedValidator Pattern Library Release 3.7 [BTC12]. These patterns act as an exhaustive reference for section 6.

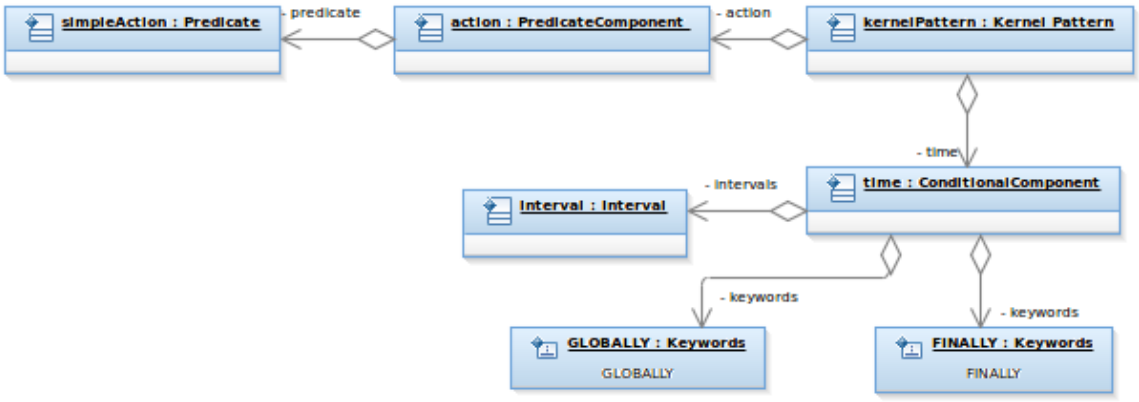

Figure 10.1: Model for finally_globally_P_B.

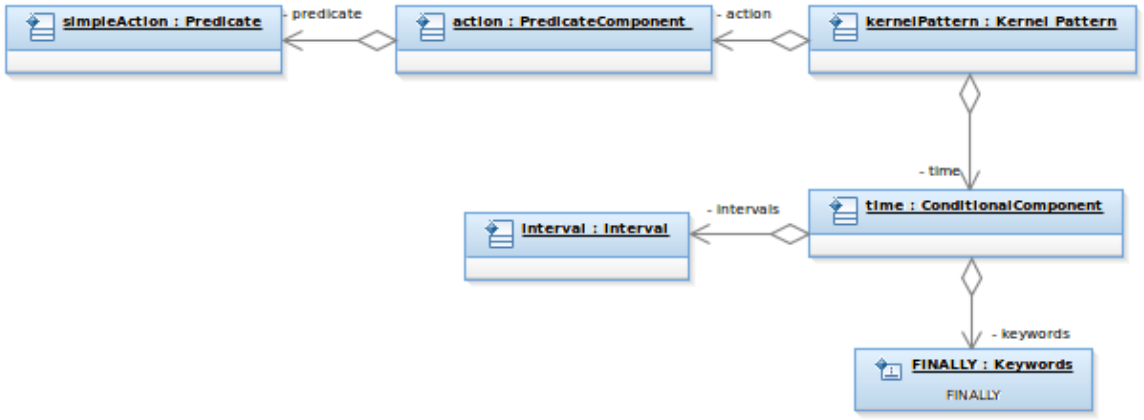

Figure 10.2: Model for finally_P_B.

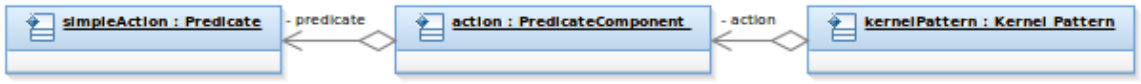

Figure 10.3: Model for P.

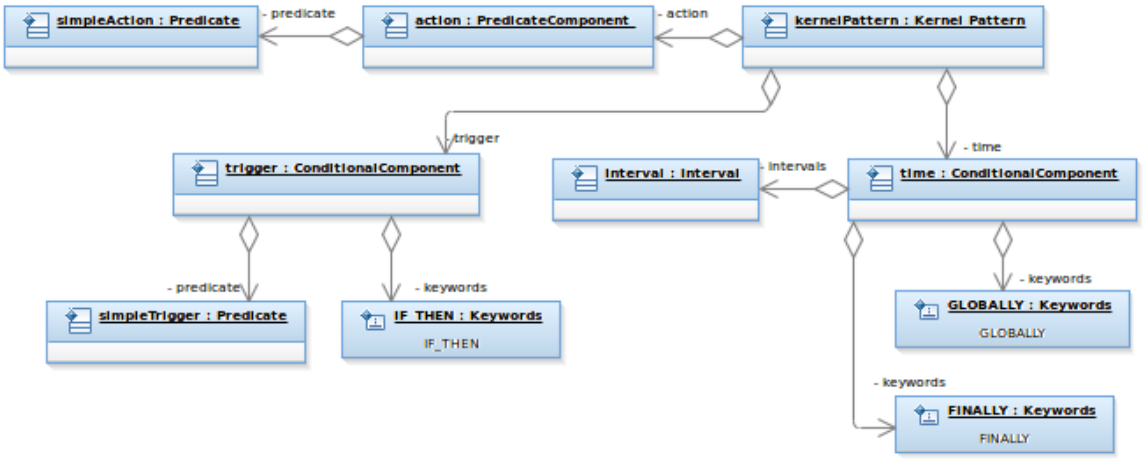

Figure 10.4: Model for P_implies_finally_globally_Q_B.

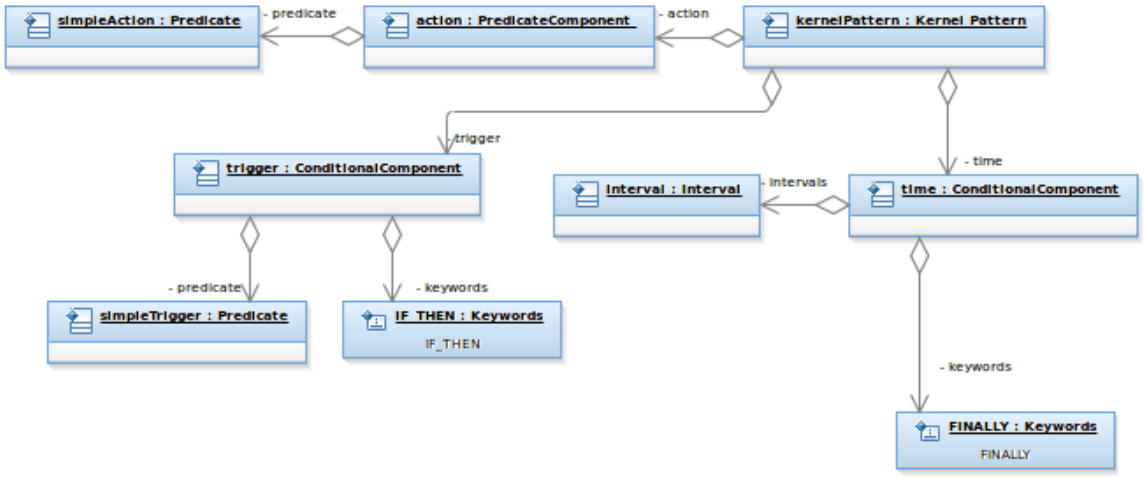

Figure 10.5: Model for P_implies_finally_Q_B.

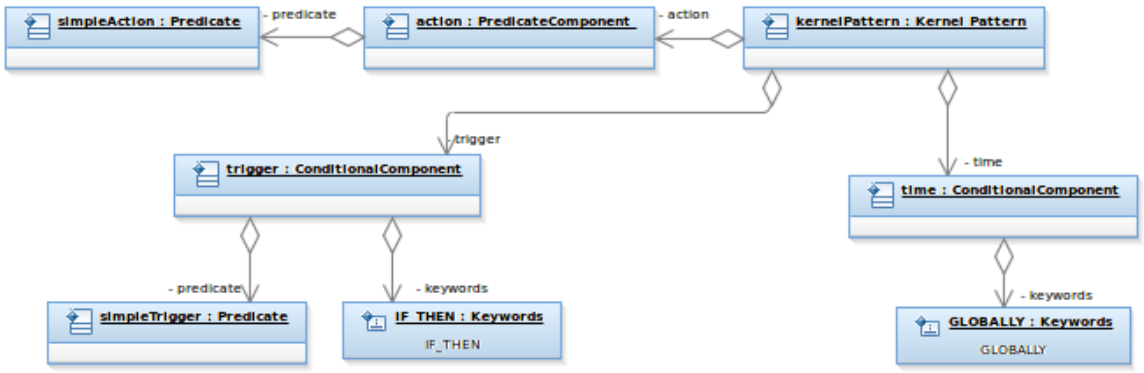

Figure 10.6: Model for P_implies_globally_Q.

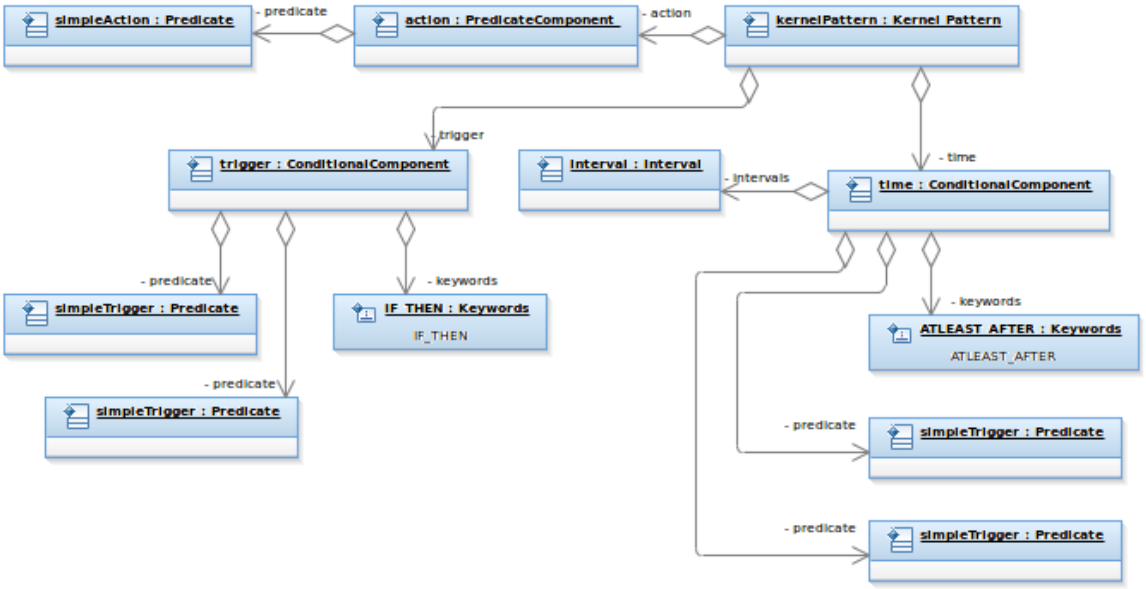

Figure 10.7: Model for P_implies_Q_atleast_X_steps_after_P.

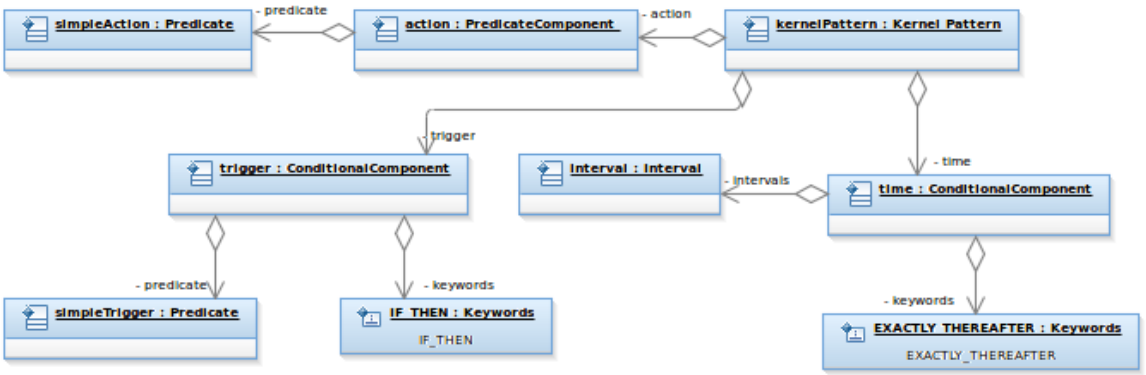

Figure 10.8: Model for P_implies_Q_at_step_X_thereafter.

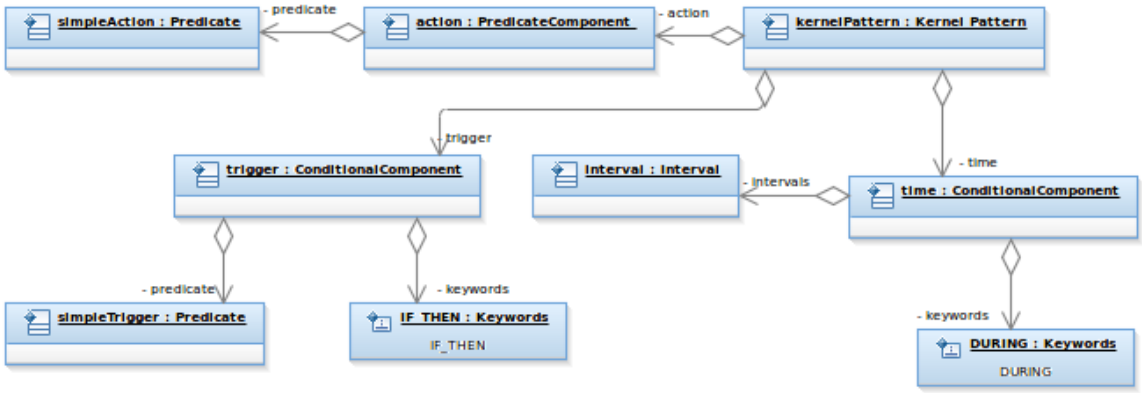

Figure 10.9: Model for P_implies_Q_during_next_X_steps.

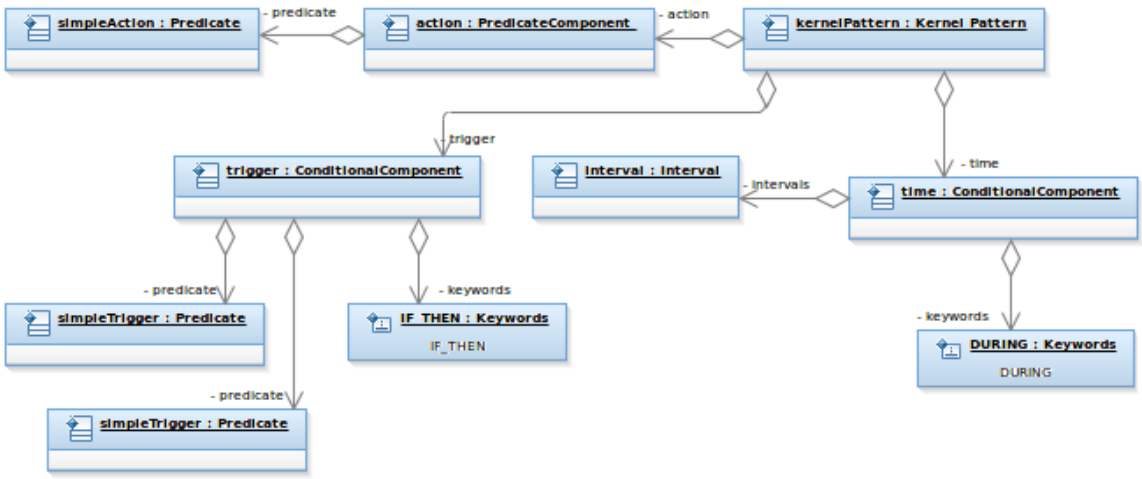

Figure 10.10: Model for P_implies_Q_during_X_steps.

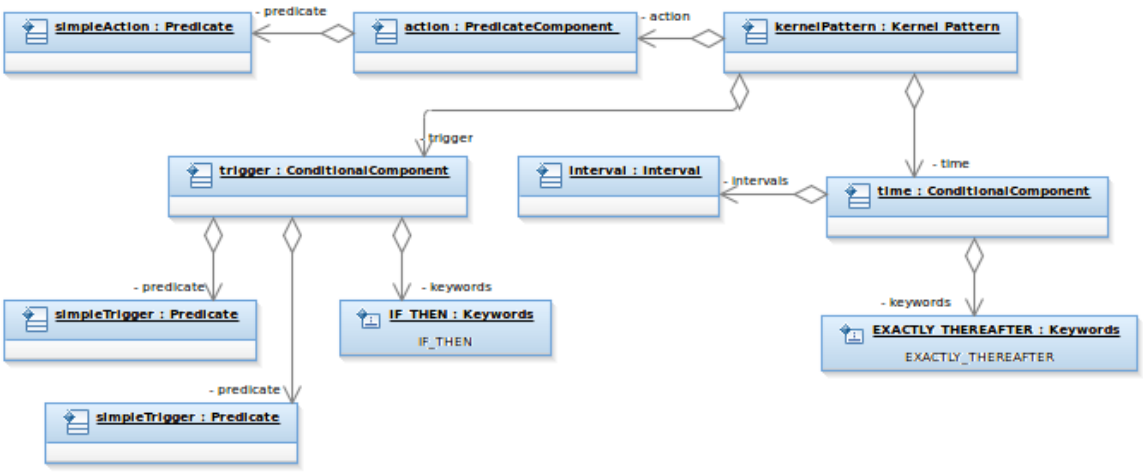

Figure 10.11: Model for P_implies_Q_X_steps_later.

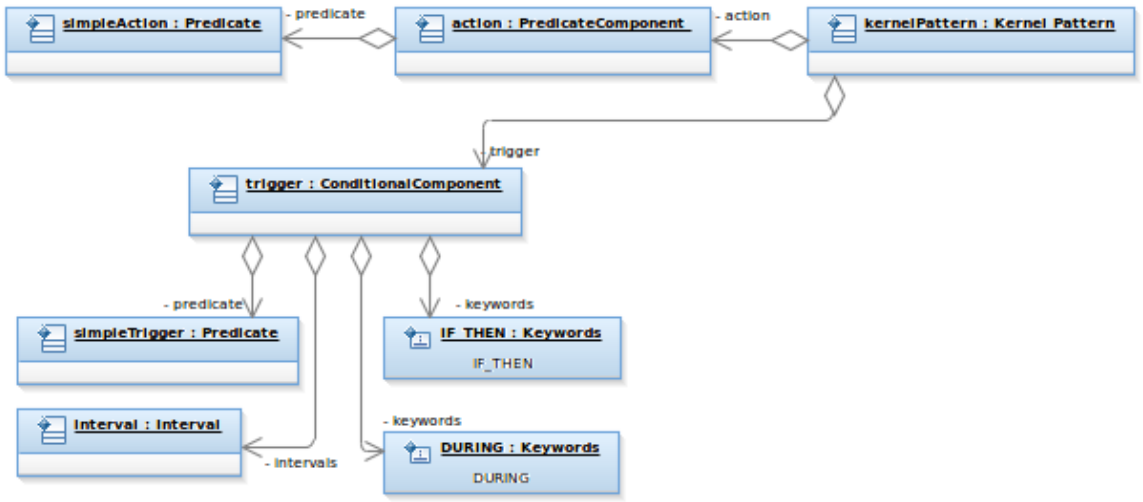

Figure 10.12: Model for P_stable_X_steps_implies_afterwards_Q.

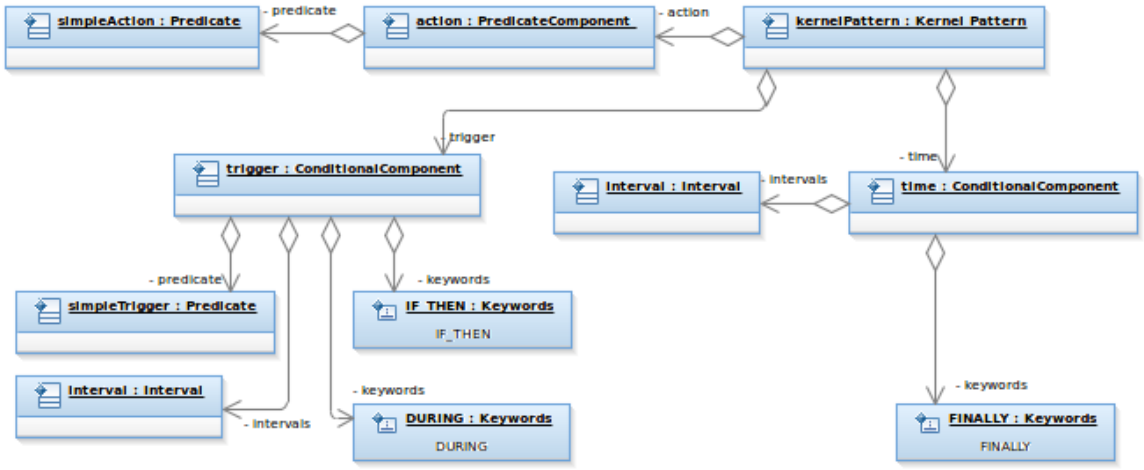

Figure 10.13: Model for P_stable_X_steps_implies_finally_Q_B.

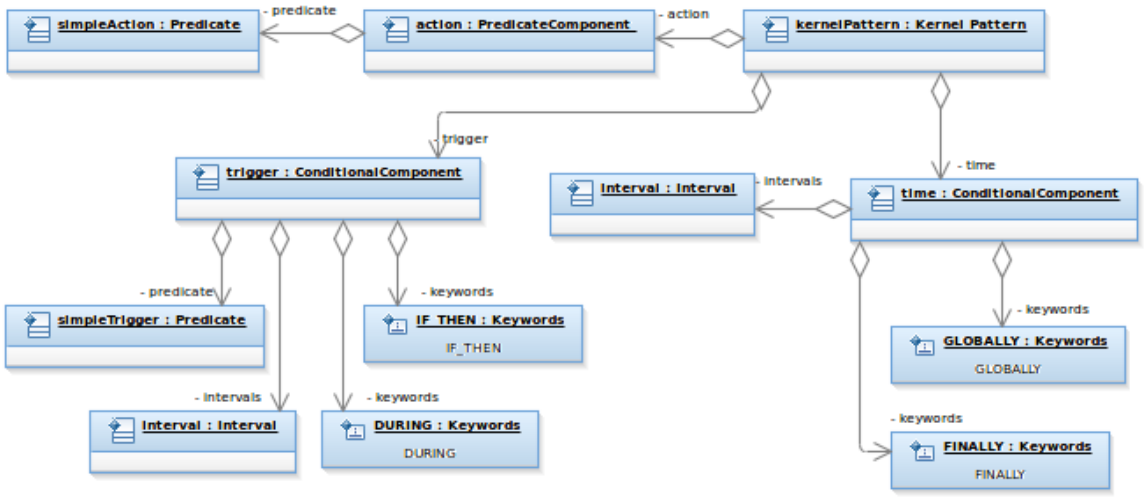

Figure 10.14: Model for P_stable_X_steps_implies_globally_Q_within_Y_steps.

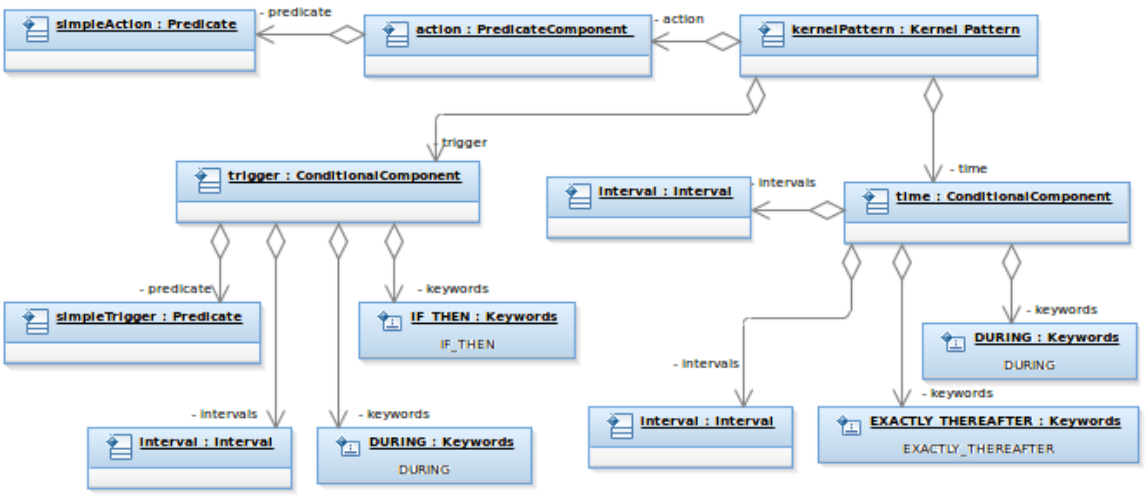

Figure 10.15: Model for P_stable_X_steps_implies_Q_stable_Y_steps_B_steps_thereafter.

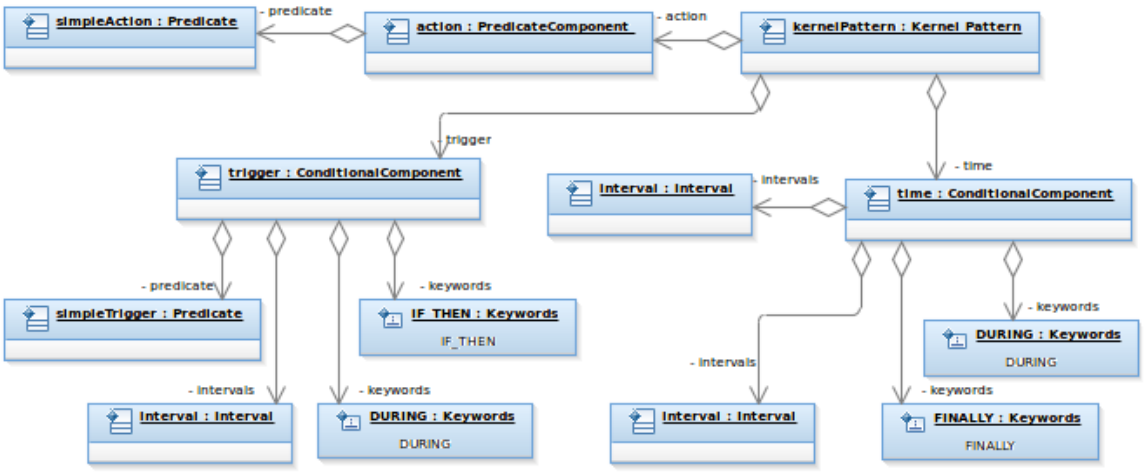

Figure 10.16: Model for P_stable_X_steps_implies_Q_stable_Y_steps_within_B_steps.

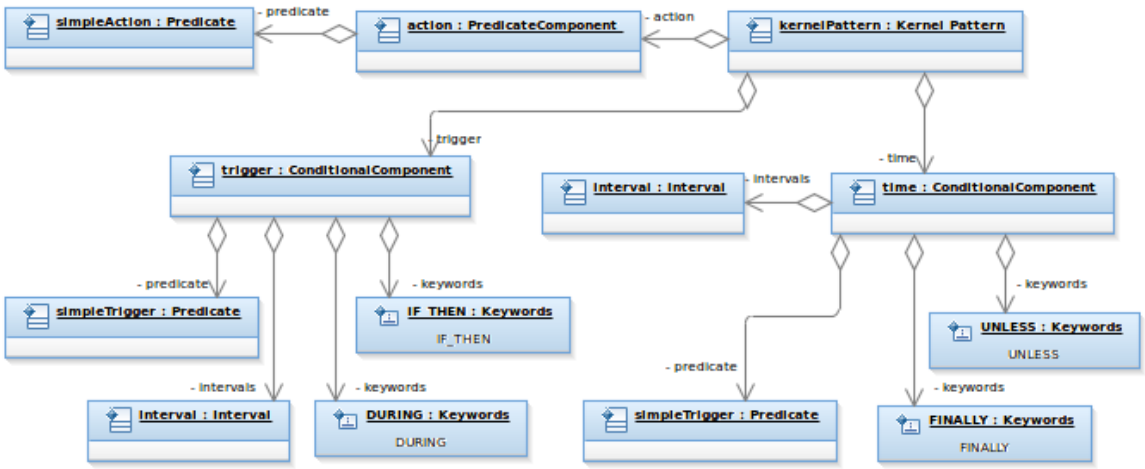

Figure 10.17: Model for P_stable_X_steps_implies_Q_within_Y_steps_unless_S.

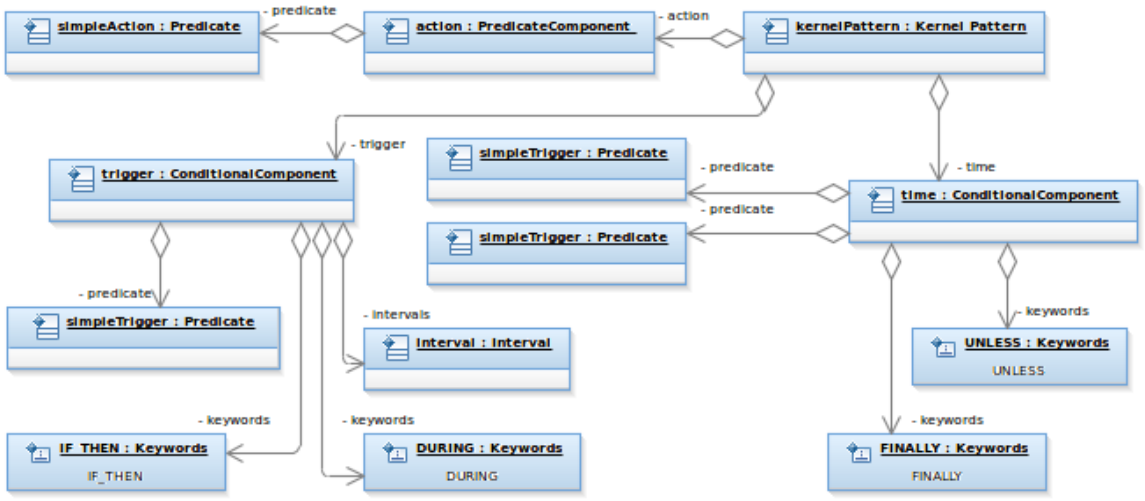

Figure 10.18: Model for P_stable_X_steps_triggers_S_releasing_Q_within_Y_steps.

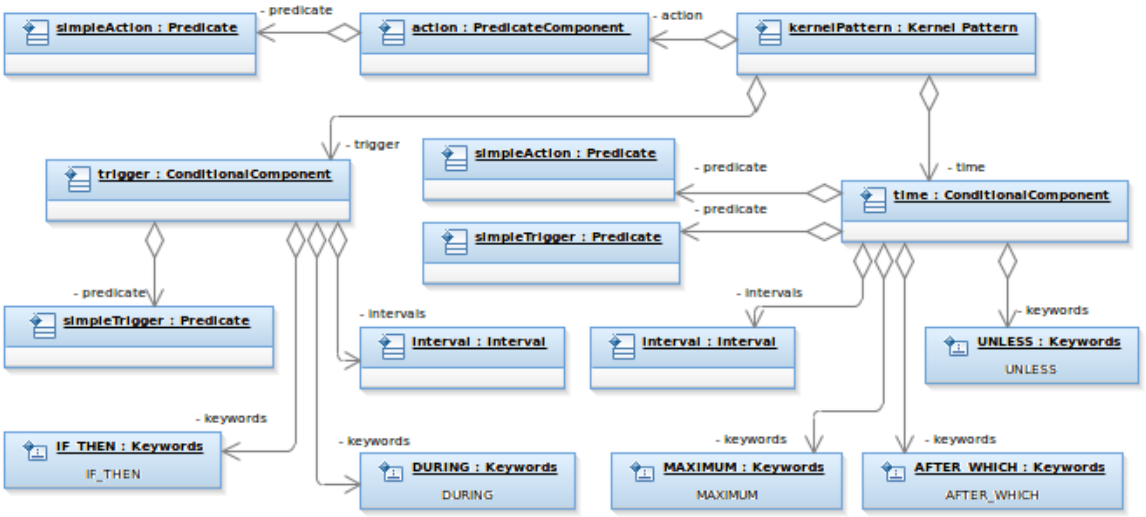

Figure 10.19: Model for P_triggering_Q_stable_X_steps_implies_S_within_Y_steps_if_stable_T.

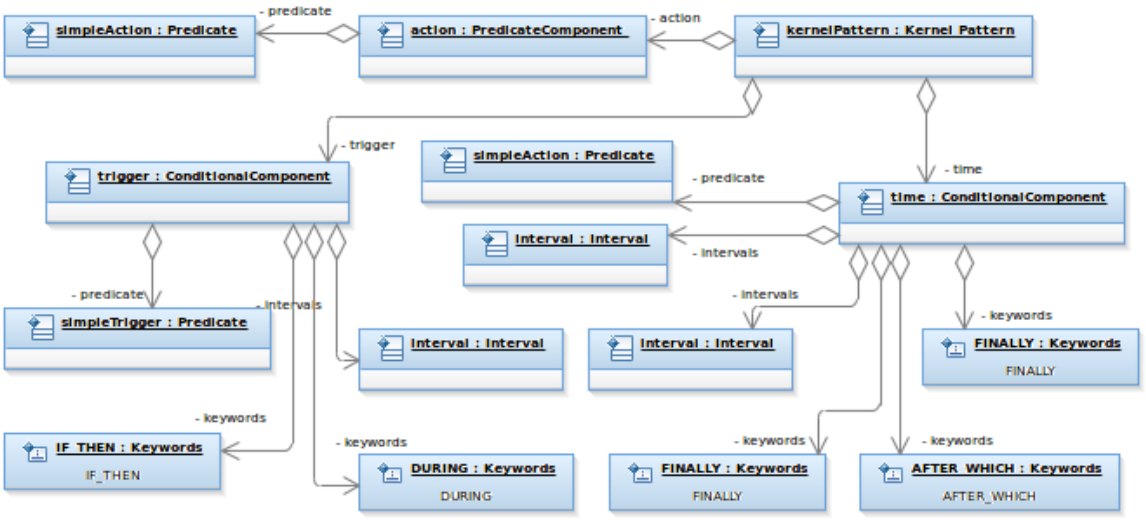

Figure 10.20: Model for P_triggering_Q_within_X_steps_implies_S_within_Y_steps.

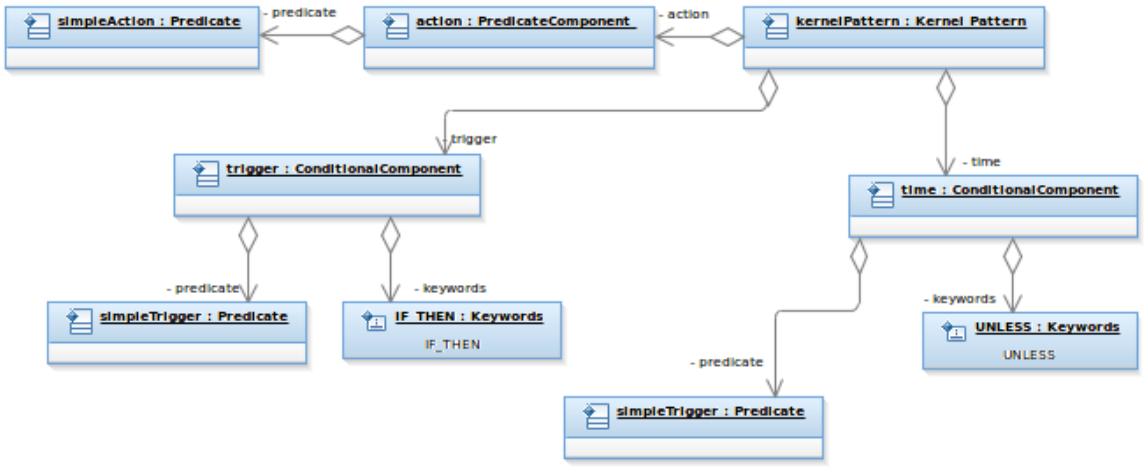

Figure 10.21: Model for P_triggers_Q_unless_S.

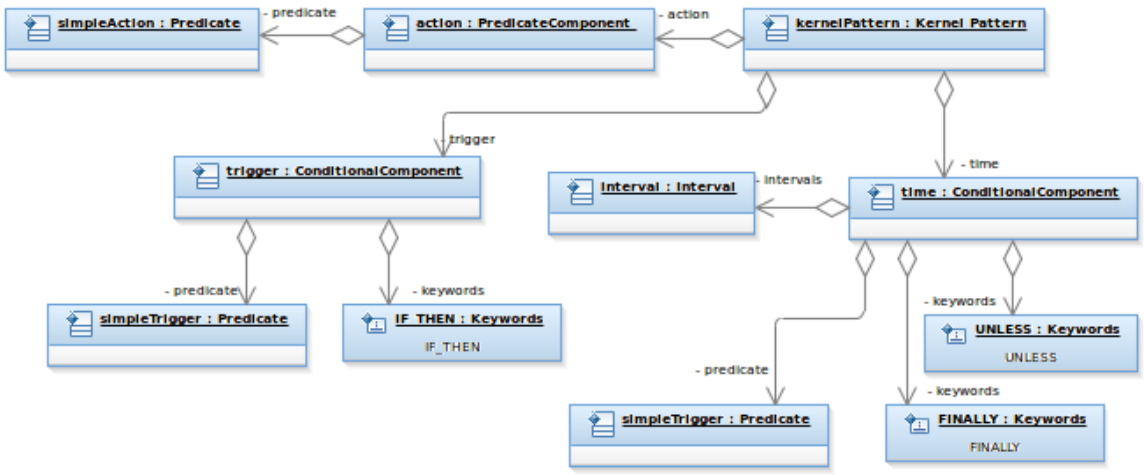

Figure 10.22: Model for P_triggers_Q_unless_S_within_B.

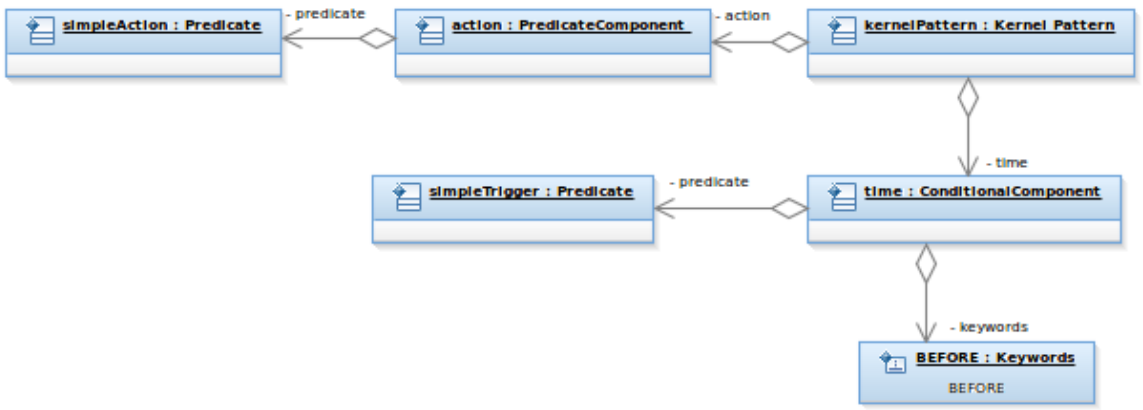

Figure 10.23: Model for Q_notbefore_P.

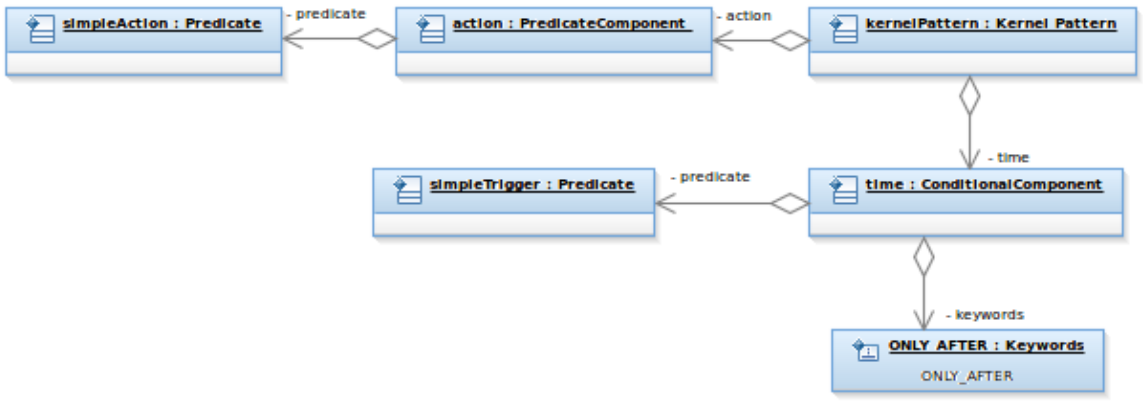

Figure 10.24: Model for Q_onlyafter_P.

CD-Contents

- **justice2013-2.pdf** – This is the thesis in portable document format.
- **JusticePrototype.jar** – This is the prototype as executable java .jar file.
- **JusticePrototype.zip** – This archive contains the netbeans project for the prototype with the following folder structure:
  - **nbproject** – contains the project files for the netbeans IDE.
  - **src** – contains the project's source files.
  - **src/justiceprototype/parser/** – contains the files for the parser.
    - **mkParser.sh** – is a shortcut to build the java files with javacc and compile them with javac.
    - **PatternParser.jj** – is the source file for the JavaCC Parser.
  - **test** – contains the JUnit tests.
  - **sanitizeParser.sh** – removes the .class files within the project. This is required before compiling the auto-generated parser .java files.

Glossary

This chapter elucidates the terms used in this document. The ∼ symbol represents a reference to the word being explained. The ↑symbol marks a term which is listed in this chapter.

Automaton An ∼ is a mathematical model, which consists of states which are connected to eachother via transitions. When it receives an input signal, it can omit an output signal and traverse a transition to change the active state. An ∼ can process computational logic.

Finite Automaton A ∼ is a type of ↑automaton which accepts input of finite length.

Stream Automaton A ∼ is an extension of a ↑finite automaton which accepts input of infinite length.

A ∼ does not necessarily stop. To determine if a ∼ accepts the input various acceptance conditions must be defined.

Büchi Automaton A ∼ is a type of ↑stream automaton. A ∼ only accepts an infinite input if at least one end state is visited infinitely often with the input.

Pattern A ∼ is an instantiation of a ↑Kernel Pattern in conjunction with a ↑start-up phase and an ↑activation mode.

Kernel Pattern A ∼ is the main component of a ↑pattern. It comprises of a ↑Trigger, an ↑Action and a ↑Temporal Constraint component.

Activation Mode The ∼ determines when a ↑kernel pattern is active. The 3 available ∼ are *initial*, *first* and *cyclic*.
initial: An *initial* ∼ defines that the ↑kernel pattern should be valid after the system was initialized.
first: A *first* ∼ defines the condition to be satisfied in order for the ↑kernel pattern to be valid if the condition occurs for the first time.
cyclic: In contrast to the *first* ∼ , the cyclic ∼ can be triggered again after the previous execution of the ↑kernel pattern is finished.

Start-up Phase The ∼ determines a precondition, before a ↑kernel pattern is active. The 3 available ∼ are *immediate*, *after n steps* and *after reaching r*
immediate: An *immediate* ∼ defines, that the ↑kernel pattern has no precondition.
after n steps: A *after n steps* ∼ defines, that the ↑kernel pattern is active n steps after system start-up.
after reaching r: A *after reaching r* ∼ defines, that the ↑kernel pattern is active after the first occurrence of r after system start-up.

Natural Language Specification A ∼ is a requirement formulated in natural human language. As the language of this document is english, all requirements said to be a ∼ are formulated in english unless explicitly stated otherwise.

Semiformal Notation Within the context of the ↑BTC EmbeddedSpecifier, a ∼ is an instanciated pattern with defined macros.

Formal Notation Within the context of the ↑BTC EmbeddedSpecifier, a ∼ is a semiformal notation, which is mapped to a target architecture, such as a TargetLink model. This mapping is done via ↑contracts.

Contract A ∼ consists of a single ↑pattern as an assurance and an arbitrary amount of ↑patterns as assumptions. If the systems state conforms to its assumptions, a contract guarantees, that the systems state also conforms to the commitment. ∼ are used to link the ↑patterns to a concrete architecture.

Macro The interface of a ↑semiformal notation consists of ∼ . ∼ are equivalent to a logical predicate. These are later mapped to signals within a ↑formal notation.

BTC EmbeddedSpecifier The ∼ is a software released in march 2013 by BTC Embedded Systems AG to simplify development with formal moethods. Due to its central role within this document, it is explained in further detail in chapter 3.

V-Model A ∼ is a software development process.

Meta-Model ∼ are defined in the UML 2.4.1 superstructure [Obj13] and is a special class diagram. A ∼ is a model of a model, thus resulting in the term ∼ .

UML ∼ stands for Unified Modeling Language and is a modeling language, which is standardized by the Object Management Group (OMG). This document complies to the UML standard version 2.4.1 [Obj13].

Requirement ∼ for software development are usually obligatory demands by the customer, which are negotiated between the customer and the developers. As software systems are complex to understand for most customers, the developers assist the customer by eliciting requirements.

Prototype A ∼ is a piece of software, which is developed as a proof of concept. A ∼ is discarded after it has fulfilled its purpose. A ∼ should never be processed into a product.

Grammar A ∼ is a set of rules, with which one can produce strings. The set of strings, which can be produced by a ∼ are called a ↑formal language.

Formal Language A ∼ is the set of strings, which can be produced by a ↑grammar.

Parser A ∼ is a program which can determine, if a given string is within a given ↑formal language. The ↑formal language, which the strings must conform to, is usually described with a ↑grammar.

Lexer See ↑Lexical Analyzer.

Lexical Analyzer A ∼ is a program, which traverses a text and replaces the plain text with ↑tokens.

Token ∼ are generated by a ↑lexical analyzer and offer a structured representation for the ↑parser.

Java ∼ is an object-oriented programming language developed by Oracle.

JavaCC ∼ is a ↑parser generator and ↑lexical analyzer, which generates ↑parsers in ↑Java.

Trigger A ∼ is a component of a ↑kernel pattern. A ∼ is a condition, which must be met, in order for the ↑action to take place.

Action A ∼ is a component of a ↑kernel pattern. A ∼ is an action, which takes place, after a ↑triggers condition is fulfilled.

Temporal Constraint A ∼ is a component of a ↑kernel pattern. A ∼ defines conditions which apply to the ↑action.

Parse Tree A ∼ is a tree, which represents the structure of a string in the context of a grammars deduction rules.

EBNF A ∼ is a structured textual representation of a grammars deduction rules. The

within this document are conform to ISO 14977 [ISO96].

Terminal Symbol A ∼ is a symbol within a ↑grammar's rule, which is deducted to a symbol, which cannot be deducted further.

Non-Terminal Symbol A ∼ is a symbol within a ↑grammar's rule, which is deducted to a symbol, which can be deducted further to another ∼ .

ISO 26262 ∼ defines obligatory requirements for the development of safety-critical systems. ∼ classifies safety critical systems into ↑ASIL-Levels depending on criteria of the development process.

ASIL-Level ↑ISO 26262 classifies safety critical systems into ∼ depending on criteria of the development process.

TargetLink ∼ is an extension to Mathworks MATLAB. It takes ↑Simulink models and generates code directly for a target architecture. It is an alternative to Mathworks Embedded Coder.

C ∼ is an imperative programming language.

C-Observer A ∼ is an end product of the BTC EmbeddedSpecifier. It is generated from formalized requirements and can verify, if the system conforms to the requirement the C-Observer was generated for.

Simulink ∼ is a module for Mathworks MATLAB, which enables model-driven development of embedded systems.

Safety Critical System ∼ are systems, which can directly endanger human lives in the case of erroneous behaviour. Among others, systems for transportation, such as cars and planes, are ∼ .

Abbreviations

AG	Aktiengesellschaft (english: joint-stock company)
ASIL	Automotive Safety Integrity Levels
BTC	Business Technology Consulting
EBNF	Extended Backus-Naur Form
EOL	End Of Line
ISO	International Organization for Standardization
JavaCC	Java Compiler Compiler
Lexer	Lexical Analyzer
RFC	Request For Comments
SUT	System Under Test
UML	Unified Modeling Language
OMG	Object Management Group

Figures

Tables

Literature

[ALSU86] AHO, A. V. ; LAM, M. S. ; SETHI, R. ; ULLMAN, J. D.: *Compilers principles, techniques, and tools*. 2nd Edition. Reading, MA : Addison-Wesley, 1986

[Bra97] BRADNER, S.: RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels. (1997), mar

[BTC] BTC ES: *BTC Specification Patterns - Nomenklatura*

[BTC12] BTC ES: *BTC Embedded Validator Pattern Library*. Release 3.7. 2012

[BTC13] BTC EMBEDDED SYSTEMS AG: *BTC EmbeddedSpecifier: Step into the formal world with ease*. `http://www.btc-es.de/index.php?lang=2&idcatside=13&mod6_26_page=1`. Version: may 2013

[FER13] FERCHAU ENGINEERING GMBH: *Funktionale Sicherheit im Zehnerpack: ISO 26262*. `http://www.ferchau.de/news/details/funktionale-sicherheit-im-zehnerpack-iso-26262-969/?ref=atFERCHAU`. Version: may 2013

[Hey13] HEYER, Sascha: *TEST DRIVEN DEVELOPMENT – EINLEITUNG*. `http://wukat.de/wordpress/2013/11/08/test-driven-development-einleitung/`. Version: November 2013

[ISO96] ISO/IEC: Information technology - Syntactic metalanguage - Extended BNF. (1996), December

[ISO12] ISO TC22/SC3: Road vehicles – Functional safety – Part 10: Guideline on ISO 26262. (2012), August

[Jav13] JAVACC PROJECT: *JavaCC [tm]: Documentation Index*. `https://javacc.java.net/doc/docindex.html`. Version: may 2013

[Joh12] JOHANNESSEN, Vegar: CESAR - text vs. boilerplates: What is more effcient - requirements written as free text or using boilerplates (templates)? (2012), august

[Nat12] NATIONAL INSTRUMENTS: What is the ISO 26262 Functional Safety Standard? (2012), February

[Nor13] NORVELL, T.S.: *The JavaCC FAQ*. `http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq-moz.htm`. Version: oct 2013

[Obj13] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language(TM) (OMG UML), Superstructure*. `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/`. Version: November 2013

[Ora13] ORACLE CORPORATION: *Variables (The Java Tutorials > Learning the Java Language > Language Basics)*. `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html`. Version: november 2013

[PP04] PERRIN, D. ; PIN, J.É.: *Infinite Words: Automata, Semigroups, Logic and Games*. Elsevier Science, 2004 (Pure and Applied Mathematics). http://books.google.de/books?id=S7hHhJc4iNgC. – ISBN 9780080525648

[PR11] POHL, K. ; RUPP, C.: *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam - Foundation Level - IREB compliant*. Rocky Nook Computing, 2011

[Rea12] REACTIVE SYSTEMS: Achieving ISO 26262 Compliance with Reactis. (2012), June

[Sla13] SLAMECKA, V.: *Information Processing - Semantic Content Analysis*. http://www.britannica.com/EBchecked/topic/444705/parsing. Version: oct 2013

[SP86] SHNEIDERMAN, B. ; PLAISANT, C.: *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Boston, MA : Addison-Wesley, 1986

Index

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den November 20, 2013

Benjamin Justice