Formalizing and Verifying Natural Language System Requirements using Petri Nets and Context based Reasoning

Aishwarya Chhabra, Amit Sangroya, C. Anantaram
TCS Innovation Labs, Gurgaon, India
{aishwarya.chhabra, amit.sangroya, c.anantaram}@tcs.com

Abstract

Natural language descriptions are generally used to describe requirements in reactive systems. Translating the natural language requirements to a more formal specification is a challenging task. One possible approach to handle complex natural language requirements is to convert them to an intermediary formal representation. This intermediate representation is further converted into a more formal representation such as EDT (Expressive Decision Tables). In this paper, we use Petri nets in combination with domain based context reasoning as a tool to model natural language requirements. We have also built a tool, NatEDT, to generate EDT specifications. In a case study, consisting of natural language requirements across three domains, our experimental results show that Petri nets provide an efficient way of formalizing natural language requirements.

1 Introduction

With the rapid growth of the internet of things (IoT), smart homes, smart cars, smart factories have become a reality. These smart systems are kind of reactive systems that interact via various sensors. It becomes a challenging task for system designers to conceptualize systems that can take complex natural language sentences as an input and test/verify the requirement. Table 1 shows some examples of natural language specifications from multiple domains of reactive systems.

The NL requirements can belong to a system as simple as *switching on a light* or as complex as *control a fire detection system remotely*. It is important for system designers to have an engineering approach to formalize NL requirement specifications. In this paper, we focus on system requirements that are primarily the descriptions of how a system (i.e. Smart Home/Automobile/IoT based system) is expected to perform in a real environment. An example of such a description in an automobile domain is as follows. *"If the ignition is on for more than 45 seconds, and seat belt is not engaged then alarm should beep"*. In this example, alarm system should beep; if the ignition is on and the seat belt is not engaged.

Many-a-times the specifications of such systems are specified in natural language sentences by designers. At times such specifications also tend to change for different system configurations and also over a period of time. In order to make it easier for system designers to define such specifications and update them, it is important to have a way to convert the specifications from natural language sentences into formal specifications. In general, requirements broadly consists of two parts: the condition part and the action part. In our approach we take NL requirement specification as an input and produce Expressive decision tables as the final output. EDT is a formal way of representing the NL specification which can be tested and verified [Venkatesh *et al.*, 2014]. EDT is a regular expression based, tabular format notation for reactive systems. An example of EDT is as follows (See Table 2).

For reactive systems, EDT representations can be easily verified for functional testing in comparison to NL specifications. This is because of the fact that natural language specifications can be ambiguous or sometimes incomplete. System designers sometimes use natural language as it can be easily written without getting into the complexity of the problem. However, to verify these reactive systems, we need test cases, and it is better to generate test cases using a formal language. Hence, it necessitates the automation of natural language to a formal language in order to bridge the gap between natural

Table 1: Example of natural language specifications

Automobile Domain: *If the ignition is on, and switch 1 is on for 2 seconds then operation 1 becomes required.*
Turn Indicator System: *When the turn indicator lever becomes left position, and the emergency flashing is off, then the flashing mode component shall assign left flashing to the flashing mode, set 0 at the flashing timer.*
Air Conditioning Domain: *When swing mode is vertical, and operation mode is cooling, then operation type becomes high speed.*

Table 2: Example of EDT (Expressive Decision Table)

in IGN	in SEAT_BELT	out ALARM
ON $\{> 45s\}$	NOT_ENGAGED	BEEP

language to test case generation.

To this end, we designed a tool called NatEDT (Natural Language to EDT) to translate the natural language specifications to formal notation using Petri nets as an intermediary and verification mechanism. Our Approach consists of several steps consisting of pre-processing, NL parsing, intermediary Petri net representation and final output as EDT. In addition to this, we have an additional step of verification that helps to test the properties like completeness and consistency of requirements. The overall approach is domain agnostic and can be easily adapted to new domains.

The remainder of this paper is structured as follows. Section 3 provides a small introduction to preliminaries which are essential for our work. Section 4 presents the proposed architecture of the system that takes natural language requirements as an input and provides a formal specification as an output. Then section 5 discusses the experimental evaluation. Section 2 offers an overview of the existing state of art approaches that focus on formalizing natural language requirement specifications. Finally, in section 6, this work ends up with some conclusions and an outlook of our future work in this area.

2 Related Work

We classify the existing approaches into two categories.

Approaches using Restrictive Natural Language
One way is to restrict the language used in writing the requirements specifications to make semantic parsing easier [Yan *et al.*, 2015]. Gutavo et al. use the Control Natural Language(CNL) for writing the specifications and have also defined a grammar for that CNL structure to do Syntactic Analysis [Carvalho *et al.*, 2014]. They also talk about the trade off between removing restrictions from the grammar and automation extent. They say that they aim at fully automating the formalizing process by restricting the language and providing a fixed format of the specifications. Since NL is controlled, a lot of manual effort is required in converting the Specification to that controlled format. Böschen et al. [Böschen *et al.*, 2016] uses a preprocessing approach to enrich the natural language requirements using a knowledge base. In this context, our approach is complementary to their work since we also use a domain ontology to contextualize the natural language requirements.

Approaches using Less Restrictive Natural Language
Less restrictive NL specifications are more natural in comparison to restrictive approaches; hence making it easier for users to write these requirements. Bajwa et al. propose an approach of scanning the specifications for relevant relationships and extracting them [Bajwa *et al.*, 2012; 2010]. They also use intermediary models from which they extract the final concepts. However, their approach is primarily based on information extraction. Other NL processing techniques such as parsing, exploiting the use of patterns, regular expressions, and rules, etc. can provide extraction of concepts and relationships from the unrestricted natural language requirement specification. Validation of the output model from the business specification must be performed which can be a laborious for large specifications.

Our approach is based upon parsing that reduces the manual effort of validating the output that is involved in approached based on information extraction [Ghosh *et al.*, 2016]. Shalini et al. proposed ARSENAL which works for less restrictive grammar but our approach verifies for the correctness using domain ontology. In this approach complete parsing is done and its semantic interpretation is done in context of the domain knowledge. Selvet et al. also takes an advantage of parsing but their approach is different in all respects as they are using SBVR (Semantics of Business Vocabulary and Business Rules) for semantic understanding [Selway *et al.*, 2015]. Sadoun et al. make the use of extracting rules automatically acquired by a training corpus, and identify concepts using a domain ontology [Sadoun *et al.*, 2013].

We also use domain knowledge in the form of ontology to validate all the identified Variables and its values. Petri nets have been used as a verification mechanism in various domains [Lee *et al.*, 2001; Sarmiento *et al.*, 2015]. Our approach is more promising as it gives an additional step of verification. Using Petri Nets as an intermediary model gives us a more robust verification mechanism and visual representation. The primary advantages of our approach over the state of art approaches is that requirements can be in less restrictive natural language. We use a domain dictionary that allows to write NL requirements using a rich vocabulary. Use of Petri nets as an intermediary helps in validation of NL specification and also preserving the context. Additionally, NatEDT has a verification process for formal verification of NL specifications.

3 Preliminaries

3.1 Expressive Decision Tables (EDT)

An EDT specification [Venkatesh *et al.*, 2014] consists of one or more tables where the column headers specify the input and output ports and the rows specify the relationship between input and output values. Each cell in a row consists of a regular expression that is used to match input streams at that port. Input values arrive as a stream at input ports at discrete time units and output values are generated as a stream at output ports at discrete time units. Example of EDT is shown in table 2, where “in” stands for input and “out” stands for output.

3.2 Ontology

In the context of knowledge sharing, the term ontology is used to mean a specification of a conceptualization. That is, an ontology is a description of the concepts and relationships that can exist for an agent or a community of agents. This definition is consistent with the usage of ontology as set-of-concept-definitions, but more general. And it is certainly a different sense of the word than its use in philosophy. We are using domain ontology which covers the concepts and their relationships with its attributes and other values. We are using *Protege* to construct the ontology in OWL format [Musen, 2015].

Domain ontology helps us specialize the approach to a particular domain which will help in fetching better results. We are using ontology for checking the identified concepts to

check if the concepts extraction does not give incorrect concepts. The verified concepts are processed further and the remaining concepts which do not belong to the domain are dropped. It also helps us identify the correct relationships of the values and the variables.

3.3 Petri Nets

Petri nets, also known as Place/Transition Nets, are used to verify work flows. Petri nets are classical models of concurrency, non-determinism, and control flow, first proposed by Carl Adam Petri in 1962. It is a collection of arcs connecting places and transitions. Places refer to the current state of the system whereas transitions are the events that take place and may cause a change in the state of the system. Places may hold tokens which enable the transitions and eventually the transition gets fired, then the tokens are distributed as per the weight given on the arcs. Places of Petri nets usually represent states or resources in the system while transitions model the activities of the system. Petri nets are bipartite graphs and provide an elegant and mathematically rigorous modeling framework for discrete event dynamically systems. Petri nets are a simple but effective method of analysing manufacturing systems [Murata, 1989; van der Aalst, 1998].

Definition 3.1. Petri Nets [Petri, 1962]

A Petri net is a four-tuple (P, T, IN, OUT) where
$P = p_1, p_2, p_3, ...p_n$ is a finite set of places
$T = t_1, t_2, t_3, ...t_n$ is a finite set of transitions
IN : is an input function that defines directed arcs from places to transitions, and
OUT : is an output function that defines directed arcs form transitions to places.

Graphically places are represented by circles and transitions represented by horizontal or vertical bars (See Figure 1). If $IN(p_i, t_j) = k$, where $k > 1$ is an integer, a directed arc from place p_i to transition t_j is drawn with the label k. If $k = 1$ we include an unlabeled arc and if $k = 0$ then no arc is drawn.

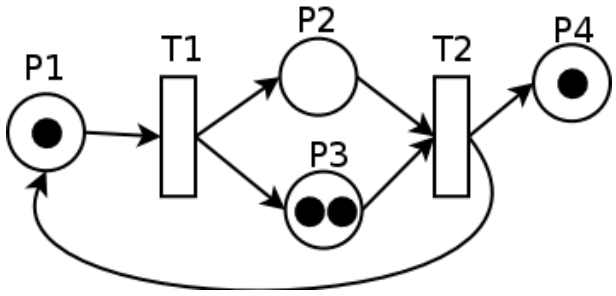

Figure 1: A simple example of Petri net

4 System Architecture

Figure 2 represents the overall architecture. System designers specify the requirement in natural language. First, we perform syntactic analysis that involves POS tagging, chunking, dependency parsing. Thereafter, we perform semantic analysis using semantic parsing techniques. We make use of a domain ontology to identify and confirm the context for a given specification. Thereafter, Petri nets are used for verification of NL specification. The output of this is a formal EDT that can be further processed for test case generation [Venkatesh *et al.*, 2016].

We designed a tool NatEDT, that takes a natural language sentence as an input and generates an equivalent EDT specification and preserves the original context. The current version of the developed prototype not only generates corresponding formal specification but also verifies the NL specifications. The overall workflow of NatEDT consist of following main steps:

4.1 Preprocessing

Each system requirement consists of mainly two parts : *condition clause* and *action clause*. In preprocessing, the first step is to split the system requirement based on its syntactic structure. The specifications when written in natural language might make use of synonyms of domain specific terms (terms present in domain ontology) instead of directly using them. To overcome this we assume that we have a domain dictionary built by domain experts. Using the domain dictionary, synonyms of the domain specific words are replaced with actual domain specific words. For example: For a given NL specification "If ignition is ON, and switch 1 is ON for 2 seconds then operation 1 becomes REQ." , this will be changed to "If IGN is ON, and SW_1 is ON for 2 seconds then OP_1 becomes REQ.". Since in this domain we consider n-gram *switch 1* as one domain term so we replace it with SW_1. Moreover, we have some general n-grams like greater than, less than or equal to. These are are replaced with greater_than, less_than_or_equal_to, respectively.

4.2 Extraction of Domain Variables

In requirement specifications, both the condition clause and action clause have some variables and values associated to these variables. We use term input variables for the terms used in conditional clause and output variables for the terms used in action clause. Extracting these variables from the requirement requires an intricate approach, which is described underneath:

Dependency Parsing

We use **Stanford CoreNLP** dependency parser for dependency parsing, which gives us a set of triples (dependent, dependency, governor) [Manning *et al.*, 2014]. For Example: (IGN, nsubj, ON) which means IGN is is related to ON and is the nominal subject for ON. We parse our specification using this Stanford typed dependency parser and get a list of such sets with various dependencies [Marneffe *et al.*, 2006]. Now let us consider the given example from general automobile domain which has time attributes also. *"If IGN is ON, and SW1 is ON for 2 seconds"*. Figure 3 represents the dependency tree for this specification.

Semantic Analysis and Validation using Ontology

Now we have all the dependencies, we can also call them grammatical relationships. We need to make sure that while

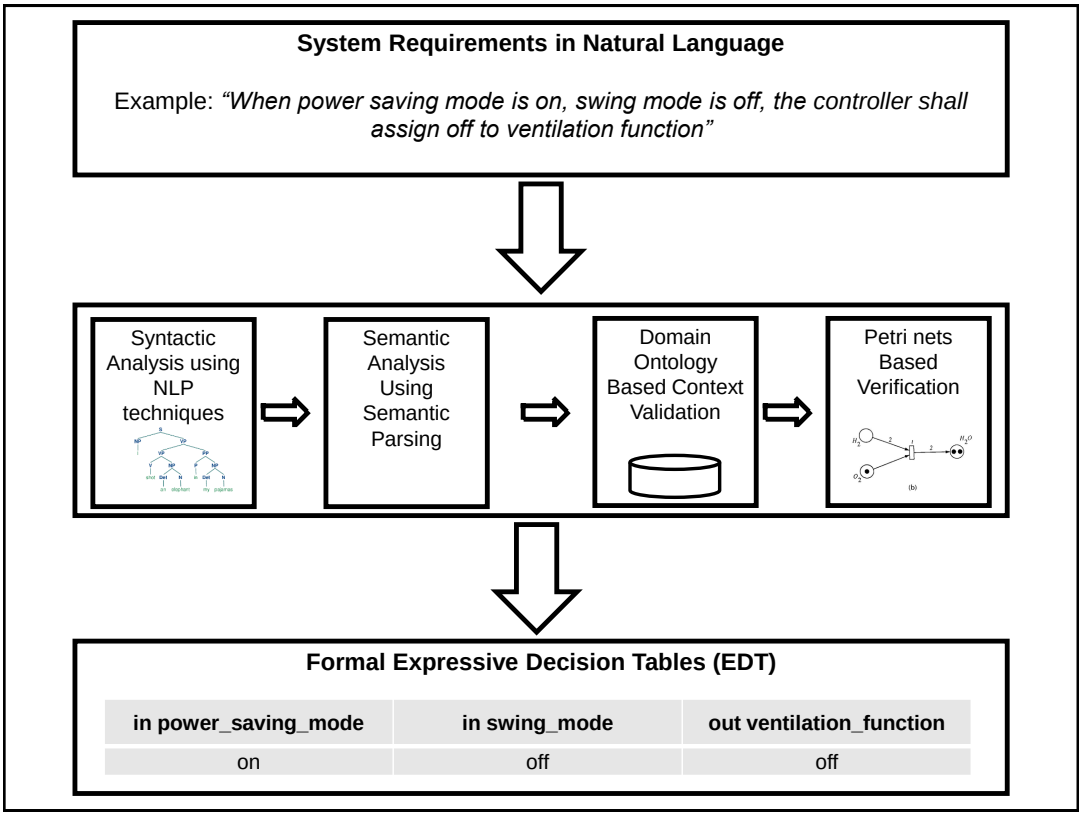

Figure 2: Overview of System Architecture

mapping them to the EDT, we don't lose the context of the specification as these are general grammar relationships. So for all the variables (concepts), we have some attributes associated to them like - value, type, time, modifier. We define some rules associated with the dependencies to fill the slots for these attributes. Using these types of rules we preserve the context of the original requirement while modeling Petri nets and generating formal EDT specifications.

For example: (SW_1, nsubj, ON) we get SW_1 is the variable and and ON is its value. We can validate the semantic relationship between SW_1 and ON using a domain ontology so that the context is preserved. (2, nummod, seconds) and (seconds , nmod:for, ON) together helps fill us the slot for time attribute. In case of "type" attribute we have two options numeric or non-numeric. For variables like voltage, timer, etc for which the value will always be numeric are categorized in that category and remaining falls into another category. We extract that using POS tags of the values. Last attribute remaining is the modifier which has the values of type greater than, less than, greater than or equal to, equal to, etc. In this example absence of any relation like this implies 'equal_to'.

4.3 Deriving Petri Nets Representations

In this step we derive Petri net based intermediate representation from the natural language requirement. To model this information into Petri nets we use python SNAKES library [Pommereau, 2008; 2015]. We assign each variable a place. The values of the variables are considered as tokens. The expression which satisfies the condition is given at the arcs. When the token fired satisfies the expression on the arcs, transition assigns tokens to the output place. Example : *"If IGN is ON, and SW_1 is ON for 2 seconds then OP_1 becomes REQ."*. In this step we can fire tokens and visualize the work flow of the requirement specification. Figure 4 shows the network before and after firing of tokens.

4.4 Generating EDT Specification

The last step is generation of EDT specifications. As initially described, EDT is in tabular format so we use python libraries like pandas to create a table for the specification. As by now we have identified the concepts and its attributes, and have also validated them in the above steps. We can map it to the EDT format. The places in the Petri nets with the tokens denotes the values at the current state of the system so we can consider places as the column names, where input place will be represented with a prefix 'in' and output variables 'out'. The tokens will be represented as the values in the corresponding rows. Figure 3 shows the table generated for the given example.

4.5 Verifying Specifications for Contextual Inconsistencies

We are verifying the extracted domain variables and the values associated to them using an ontology in the transformation process itself. We have added an additional verification step to to the tool which verifies the other requirements

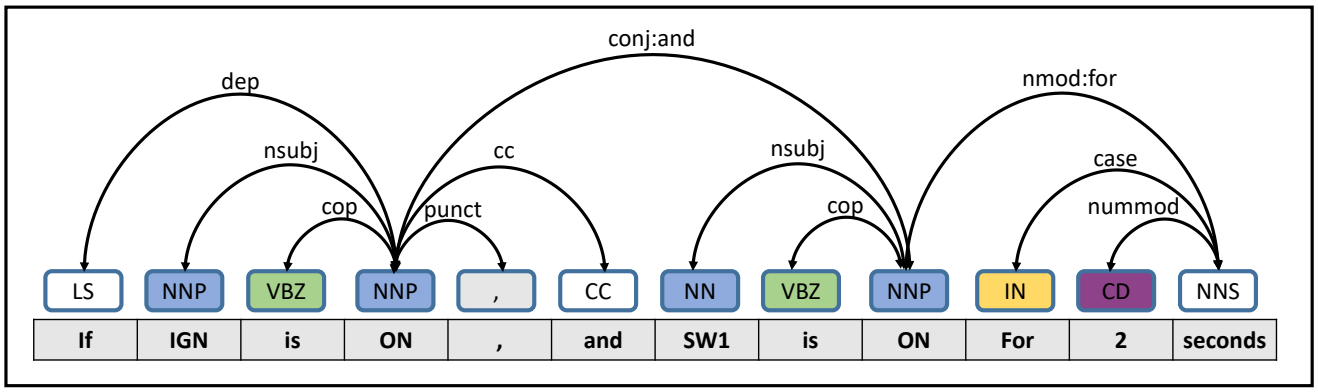

Figure 3: Dependency Graph for NL Specification

Table 3: EDT for the representative example

in IGN	in SW_1	out OP_1
ON	ON	REQ

given to check their consistency with the existing requirements. We process a finite number of specifications S (where $S = S_1, S_2, S_3, ...S_n$) as explained in the above sections and store the information extracted at each step in a file. When a user provides a new input condition (C), it is processed as explained in previous sections. The information extracted from C is matched to the information extracted for specifications $(S_1, S_2, S_3, ...S_n)$ using the verification algorithm which returns complete specification with a Petri net for the given input conditions if an exact match is found. If the exact match is not found it looks for the best match which refers to the one with the highest number of matching input conditions and returns the Petri net and table stored for those specifications.

For Example: We originally had a specification in set S : *"If screen is unlock and power button is pressed for less than 1 second and released, then turn screen to lock"*. User describes an input condition as: "If screen is unlock and power button is pressed for less than 3 second and released", the tool process this condition and extract two input variables: **SCREEN** having a value **UNLOCK** and **POWER_BUTTON** having a value **PRESSED** for less than 3s and then changes to **RELEASED**. The verification algorithm couldn't find an exact match in set S, hence looks for the best match highlighting the differences. In this example, it highlights the time attribute is different from the existing best match for the given input. Thereafter, it provides user an additional option to either correct if it was an error or add it as a new requirement specification in the set S by providing the output action for the corresponding input.

5 Experimental Evaluation

NatEDT tool was tried on two different set of requirements from automobile domain (**Turn Indicator Systems** (17 requirements) and another automobile sample set [Venkatesh *et al.*, 2014] (29 requirements). We also tested toy examples from **Air Conditioner** domain (See table 5). We tested

Table 4: Precision and Recall for various domains

	# Samples	Recall %	Precision%
Turn Indicator System	76	85.52%	100%
Automobile	49	91.78%	94.36%
Air Conditioning	17	100%	100%

on different domains to test the adaptability of the approach on different domains and we realized that the approach is generic. To adapt to different domains one need to have external domain dictionary for preprocessing and domain ontology for verification of those specifications. Our evaluation criteria was based upon calculating the precision and recall.

We calculated the total number of concepts that need to be identified in each sample set and then the variables which were correctly identified, incorrectly identified, and the missed concepts. The largest requirement in english was composed of 48 words and the smallest sentence was composed of 12 words. The TIS sample was mostly state based examples and the other set had some state based as well as examples having time and stream of inputs. The results are compiled in table 4. We get a precision of 94.36% and a recall of 91.78% in automobile domain. Though we cannot compare our results with any other work as the formal notation and approach used in our paper is quite different than the formal notations and approaches in prior work. Our results are quite promising for the transformation to a relatively new formal notation.

6 Conclusions and Future Work

To this end, a tool called NatEDT is developed that takes a natural language sentence as an input and generates an EDT specification. We make use of domain knowledge in the form of dictionary and ontology to preserve the context in NL specification. We are also able to verify the new NL requirements based on the existing requirements for its consistency and completeness. In the future, this work could be extended for robust verification and validation of the requirements in the system.

Table 5: Examples of Natural Language Specification and their corresponding EDTs

1	If Mist Remover Switch is on and Mist Remover controller is NO_req then Mist Remover Request becomes on and Mist remover relay is on.	in Mist_Remover_SW	in Mist_Remover_Ctrl	out Mist_Remover_Request	out Mist_Remover_Relay
		ON	NO_REQ	ON	ON
2	If Mist Remover Switch is off and Mist Remover controller is NO_req then Mist Remover Request becomes off and Mist remover relay is off.	in Mist_Remover_SW	in Mist_Remover_Ctrl	out Mist_Remover_Request	out Mist_Remover_Relay
		OFF	NO_REQ	OFF	OFF
3	If alarm is ON, and panicsw is pressed for more than 3 seconds and released then the flash should be NO_REQ, and alarm should be OFF.	in alarm	in panicSw	out flash	out alarm
		ON	pressed $> 3s$ released	NO_REQ	OFF
4	When the turn indicator lever becomes left position, and the emergency flashing is off, then the flashing mode component shall assign left flashing to the flashing mode, reset the flashing timer.	In turn_indicator_lever	In emergency_flashing	Out flashing_mode	Out flashing_timer
		Left_position	off	right_flashing	0
5	When the emergency flashing is off, and the turn indicator lever becomes the right position, and the flashing mode is not right flashing, the flashing mode component shall assign right flashing to the flashing mode, reset the flashing timer.	In turn_indicator_lever	In emergency_flashing	Out flashing_mode	Out flashing_timer
		Right_position	on	left_flashing	0

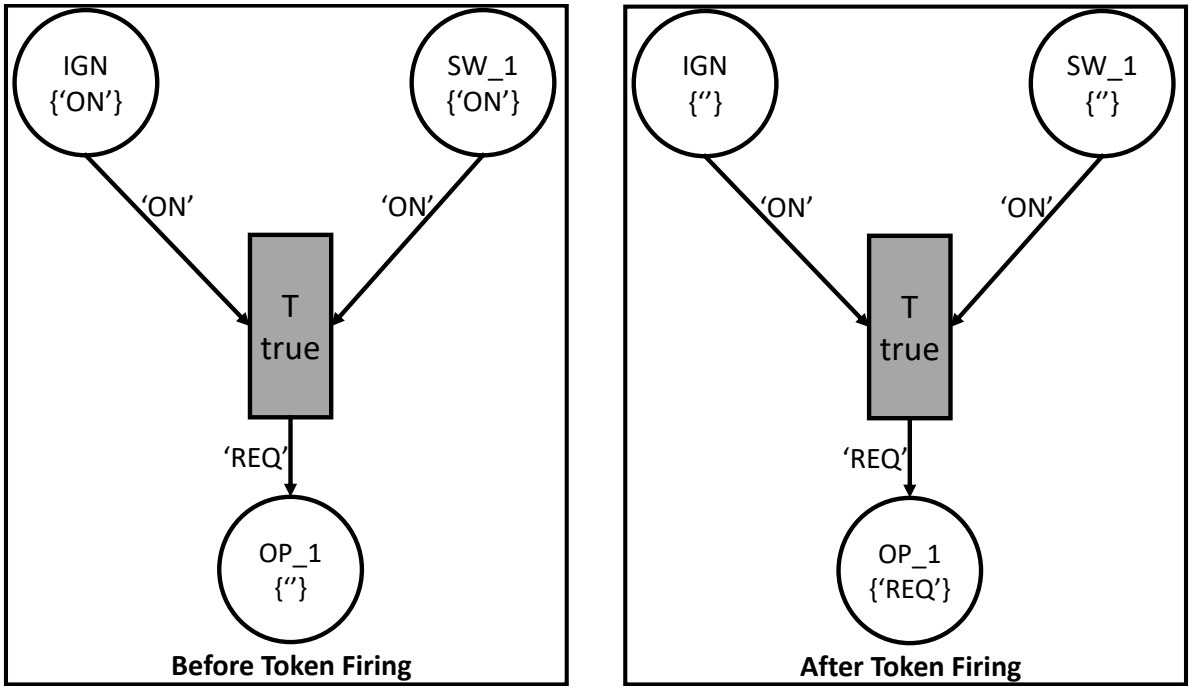

Figure 4: Before and After Firing of Tokens

References

[Bajwa *et al.*, 2010] I. S. Bajwa, B. Bordbar, and M. G. Lee. Ocl constraints generation from natural language specification. In *2010 14th IEEE International Enterprise Distributed Object Computing Conference*, pages 204–213, Oct 2010.

[Bajwa *et al.*, 2012] Imran Bajwa, Behzad Bordbar, and Mark Lee. Nl2alloy: A tool to generate alloy from nl constraints. 10:365–372, 12 2012.

[Böschen *et al.*, 2016] Martin Böschen, Ralf Bogusch, Anabel Fraga, and Christian Rudat. Bridging the gap between natural language requirements and formal specifications. In *Joint Proceedings of REFSQ-2016 Workshops, Doctoral Symposium, Research Method Track, and Poster Track co-located with the 22nd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2016), Gothenburg, Sweden, March 14, 2016.*, 2016.

[Carvalho *et al.*, 2014] Gustavo Carvalho, Ana Carvalho, Eduardo Rocha, Ana Cavalcanti, and Augusto Sampaio. A formal model for natural-language timed requirements of reactive systems. In Stephan Merz and Jun Pang, editors, *Formal Methods and Software Engineering*, pages 43–58, Cham, 2014. Springer International Publishing.

[Ghosh *et al.*, 2016] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. Arsenal: Automatic requirements specification extraction from natural language. In Sanjai Rayadurgam and Oksana Tkachuk, editors, *NASA Formal Methods*, pages 41–46, Cham, 2016. Springer International Publishing.

[Lee *et al.*, 2001] Jonathan Lee, Jiann-I Pan, and Jong-Yih Kuo. Verifying scenarios with time petri-nets. *Information and Software Technology*, 43(13):769 – 781, 2001.

[Manning *et al.*, 2014] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[Marneffe *et al.*, 2006] M. Marneffe, B. Maccartney, and C. Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC-2006)*, Genoa, Italy, May 2006. European Language Resources Association (ELRA). ACL Anthology Identifier: L06-1260.

[Murata, 1989] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Musen, 2015] Mark A. Musen. The protege project: A look back and a look forward. *AI Matters*, 1(4):4–12, June 2015.

[Petri, 1962] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universitat Hamburg, 1962.

[Pommereau, 2008] Franck Pommereau. Quickly prototyping Petri nets tools with SNAKES. *Petri net newsletter*, (10-2008):1–18, 10 2008.

[Pommereau, 2015] Franck Pommereau. SNAKES: a flexible high-level Petri nets library. In *Proceedings of*

PETRI NETS'15, volume 9115 of *LNCS*, pages 254–265. Springer, 06 2015.

[Sadoun *et al.*, 2013] D. Sadoun, C. Dubois, Y. Ghamri-Doudane, and B. Grau. From natural language requirements to formal specification using an ontology. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 755–760, Nov 2013.

[Sarmiento *et al.*, 2015] E. Sarmiento, J. C. S. D. P. Leite, and E. Almentero. Analysis of scenarios with petri-net models. In *2015 29th Brazilian Symposium on Software Engineering*, pages 90–99, Sept 2015.

[Selway *et al.*, 2015] Matt Selway, Georg Grossmann, Wolfgang Mayer, and Markus Stumptner. Formalising natural language specifications using a cognitive linguistic/configuration based approach. 54, 04 2015.

[van der Aalst, 1998] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[Venkatesh *et al.*, 2014] R. Venkatesh, U. Shrotri, G. M. Krishna, and S. Agrawal. Edt: A specification notation for reactive systems. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[Venkatesh *et al.*, 2016] R. Venkatesh, Ulka Shrotri, Amey Zare, and Supriya Agrawal. On generating test cases from edt specifications. In Leszek A. Maciaszek and Joaquim Filipe, editors, *Evaluation of Novel Approaches to Software Engineering*, pages 1–20, Cham, 2016. Springer International Publishing.

[Yan *et al.*, 2015] R. Yan, C. H. Cheng, and Y. Chai. Formal consistency checking over specifications in natural languages. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1677–1682, March 2015.