Ontology-Driven Natural Language Requirement Templates for Model Checking I&C Functions

Teemu Tommila, Antti Pakonen, Janne Valkonen
VTT Technical Research Centre of Finland
+358 20 722 111, firstname.lastname@vtt.fi

Abstract

Model checking is a formal and exhaustive computer-aided verification method particularly suitable for I&C functions. A challenge with it is that both requirements and I&C functions must be first transformed into a formal presentation understood by the model checker software. Collecting and formalizing the requirements requires domain expertise and ability to handle the sometimes tricky formal presentations. In this paper, we suggest a combination of ontologies and natural language templates as a means to tackle this problem. On this basis, an intelligent text editor can be implemented to guide the user in writing well-formed requirements. Storing templates together with their formal counterparts makes it possible to automatically formalize the requirements for the model checker.

1. Introduction

Requirements are the basis of designing and evaluating technical systems. In critical applications, such as nuclear power plants, safety must be demonstrated with argumentation and traceable documentation from the design bases and regulatory requirements to the implementation. In large facility investments, it is also important that safety requirements are correctly communicated to all parties in the distributed project organisation. Therefore, requirements must be technically correct, easily understandable, unambiguous and verifiable.

Unfortunately, Requirements Engineering is still poorly understood and implemented in the industry [4]. Requirements are typically defined informally in written form. Engineers tend to use complex sentences, inconsistent terminology and words with a vague meaning. Requirements become difficult to verify, hard to read and have several interpretations. Requirements are often edited using regular office tools that are isolated from the primary engineering environments and have no decent support for traceability and information management. To tackle these problems, designers should be provided with support for creating and managing high-quality requirements.

In this paper, we suggest a combination of two techniques, namely ontologies as a way towards a precise terminology and natural language templates as a means to improve the structure of written requirements. The ontology represents the concepts of the application domain and sets rules for inserting individuals into the knowledge base. In the templates, the concepts of the ontology are used as placeholders that will be replaced by application-specific individuals in the requirement statements. The ontology and the templates together define a simple Controlled Natural Language (CNL) with a restricted vocabulary and grammar.

We use this approach for model checking, which is a formal method suitable for verifying safety-critical systems. One of its challenges is, however, that the requirements must be rewritten with the formal language used by the model checker software. The approach that we suggest here is to define the requirement templates together with their corresponding presentations in temporal logic.

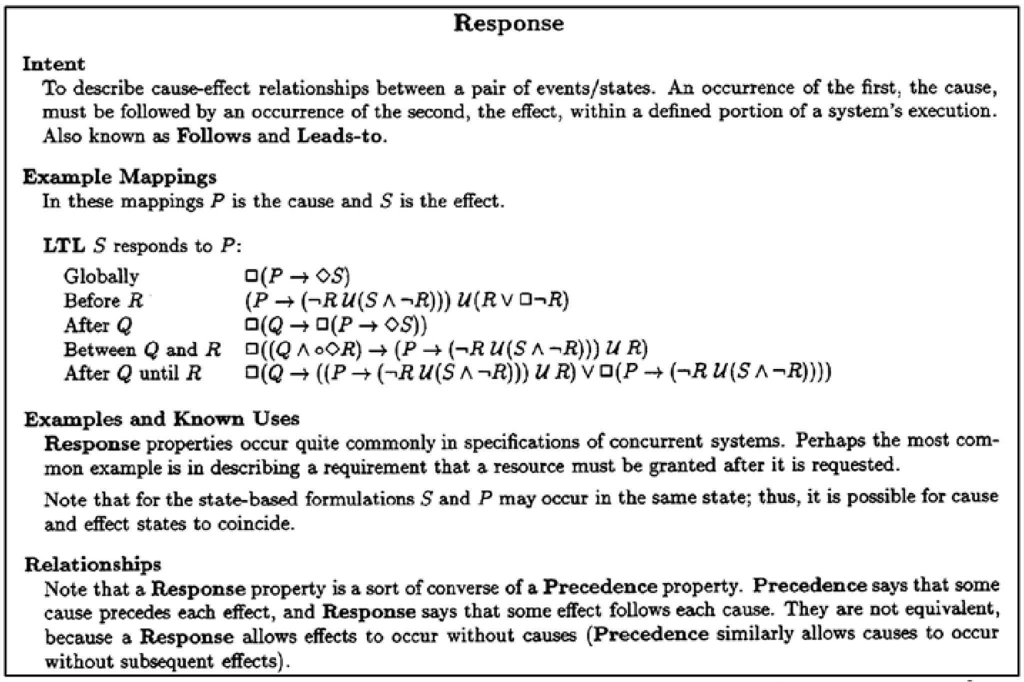
On the basis of the templates and the domain-specific ontology, we have implemented a demonstration tool that guides the user in writing well-formed requirements and exporting them to the model checker.

2. Related research

CNL is a widely used approach to improve document readability and to allow computer processing of textual data. Also requirement templates are an old idea found in textbooks, standards and even in commercial software tools. For example, ISO 29148 [1] gives guidance on well-formed requirement statements. Mavin et al. [2] have defined small set of generic templates for functional requirements. Boilerplates are an old example of the approach originally suggested by Hull, Jackson & Dick [3] (http://www.requirementsengineering.info). Boilerplates contain a larger number of templates and is also integrated as a plugin to the requirements management system DOORS. The idea is quite simple - choose a template and fill in the gaps, for example in the sentence

While <operational condition> the <user> shall able to <capability>.

The combination of ontologies and natural language processing is considered a promising way to better requirements authoring and verification, and many tools are available for that [4]. Requirement templates have been applied in the design and analysis of safety-critical (control) systems also. For example, Stålhane et al. [5] describe an approach to functional safety analysis during the early development phases. The prototype tool developed in the European funded CESAR project uses boilerplates to organize the requirements and an ontology to model domain entities and their failure modes.



Response

Intent

To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect, within a defined portion of a system's execution. Also known as **Follows** and **Leads-to**.

Example Mappings

In these mappings P is the cause and S is the effect.

LTL S responds to P:

Globally	$\Box(P \rightarrow \Diamond S)$
Before R	$(P \rightarrow (\neg R\, \mathcal{U} (S \wedge \neg R)))\, \mathcal{U} (R \vee \Box\neg R)$
After Q	$\Box(Q \rightarrow \Box(P \rightarrow \Diamond S))$
Between Q and R	$\Box((Q \wedge \circ\Diamond R) \rightarrow (P \rightarrow (\neg R\, \mathcal{U} (S \wedge \neg R)))\, \mathcal{U}\, R)$
After Q until R	$\Box(Q \rightarrow ((P \rightarrow (\neg R\, \mathcal{U} (S \wedge \neg R)))\, \mathcal{U}\, R) \vee \Box(P \rightarrow (\neg R\, \mathcal{U} (S \wedge \neg R))))$

Examples and Known Uses

Response properties occur quite commonly in specifications of concurrent systems. Perhaps the most common example is in describing a requirement that a resource must be granted after it is requested.

Note that for the state-based formulations S and P may occur in the same state; thus, it is possible for cause and effect states to coincide.

Relationships

Note that a **Response** property is a sort of converse of a **Precedence** property. **Precedence** says that some cause precedes each effect, and **Response** says that some effect follows each cause. They are not equivalent, because a **Response** allows effects to occur without causes (**Precedence** similarly allows causes to occur without subsequent effects).

Figure 1. The "Response" property specification pattern helps in expressing with temporal logic that a certain state S must follow the state P. (Modified from [6]).

The idea of reusable templates is also applied in formal property specification patterns used in verification tools such as model checkers [6]. An oft-cited collection of such patterns can be found in [7]. The patterns aim at capturing common, recurring requirement expressions in terms of languages such as temporal logic. The user is provided with a natural language description of a requirement along with the mapping to temporal logic (see Figure 1 for an example). A requirement specification tool based on similar patterns called IFADIS is introduced in [8]. An example of the use of property specification patterns in the context of modelling cellular information networks can be found in [9].

3. Model checking

3.1 Model checking as a verification method

Model checking [10] is a computer-assisted method for verifying the behaviour of a (hardware or software) system model against a set of requirements. Both the system and its requirements are modelled with a formal language. A software tool called a “model checker” can then examine, whether all possible behaviours of the model fulfil the specified requirements. If a behaviour is found that is contrary to a requirement, a counter-example demonstrates the unwanted scenario. By reviewing the counter-example, solid evidence of a design error can be found.

The computational power of the algorithms involved makes model checking an effective tool in real-world applications. The key benefit over other V&V methods (such as simulation or testing) is that the analysis is exhaustive – all the possible states and executions of the system model are taken into account. Exhaustive analysis is possible because model checking is not a brute force method. Only those model behaviours that are relevant for the stated requirement are processed.

Since the 1990’s, model checking has been applied in domains such as microprocessor manufacturing, and later in diverse applications ranging from aviation and space to the verification of Windows device driver source code. At VTT, we have developed methods and tools for model checking of I&C software, and also put them into practical use [11]. For example, we have been consulting both the Finnish Radiation and Nuclear Safety Authority (STUK) and the power company Fortum on evaluating nuclear I&C systems [12]. In addition to the nuclear domain, we have conducted several pilots in the fields of factory automation, conventional power plants, and electrical and machine automation.

3.2 Model checking as a work process

The overall process of performing model checking is illustrated in Figure 2. While the actual analysis is performed automatically by the model checker, there are several tasks that require manual work and specific expertise. Although there are versatile model checkers available, the tools are either completely generic, or tailored to some specific domain, and therefore not directly applicable for I&C software verification.

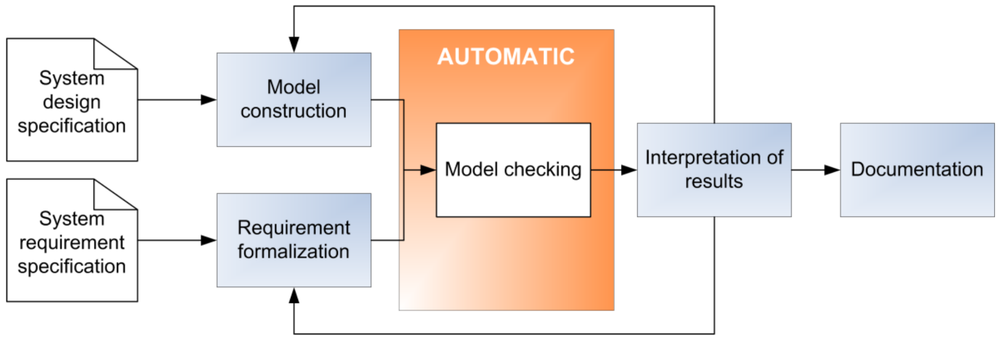

System design specification
System requirement specification
Model construction
Requirement formalization
AUTOMATIC
Model checking
Interpretation of results
Documentation

Figure 2. The overall model checking process involves several steps where manual work and specific expertise are required. In this paper, we focus on requirement formalisation.

The system model is constructed on the basis of design documentation, e.g., function block or logic diagrams. Typically, a state machine representation is used. The modeller will usually have to resort to simplifications or abstractions, and expertise is needed to correctly describe all relevant aspects of system behaviour.

The functional I&C requirements are formalised using temporal logic for expressing the correct model behaviour over time. This step can be difficult because the original (natural language) requirements can be vague, ambiguous or incomplete, and because of the complexity of temporal logic languages (see section 3.4).

The model checker will then either state that a given requirement holds, or return a counter-example. The review of the counter-example can reveal 1) an actual design error, 2) an error in the way the model was constructed, or 3) an inaccurately formulated requirement. In the latter cases, suitable refinements are made, and the analysis is performed again. Model checking can therefore be said to repair modelling errors to some extent. Still, the user must typically deal with a large amount of counter-examples. So, the review and iteration process may be very laborious due to the large amount of data and poor visualisation provided by current model checkers.

3.3 Model checking of systems based on function blocks

When verifying I&C systems based on the function block paradigm, we have found that the modelling task can be made more efficient by adopting a modular approach. We first construct a library of model checker code elements that corresponds to the set of (vendor-specific) function block types. Then, the model for the application logic can be constructed by simply connecting the code blocks in a way similar to wiring ports of function blocks. Although some aspects still require special care (e.g., timing, discretization of analogue signals, asynchrony), the overall task is easier and less prone to errors.

The modular approach also enables us to develop graphical tools for model checking [12]. Our solution is based on Simantics, an open source platform for modelling and simulation [13, 14]. Based on a semantic modelling kernel, Simantics enables the connection and co-use of many simulation and engineering tools – OpenModelica, BALAS, Apros, OpenFoam, Comos and SmartPlant, to name a few. For model checking, Simantics provides a graphical modelling framework that enables us to:

- Construct the model by instantiating and connecting block elements in a 2D graphical view (see Figure 3). Eventually, we expect that the model translation capabilities of Simantics will enable automatic model conversion from an existing I&C specification.
- Generate an input file for the model checker, in our case NuSMV [15], and run the analysis.
- Review the counter-examples produced by NuSMV as trends or animations of a “living” function block diagram (showing how signal values change over time by e.g. changing the colours of the block connections).

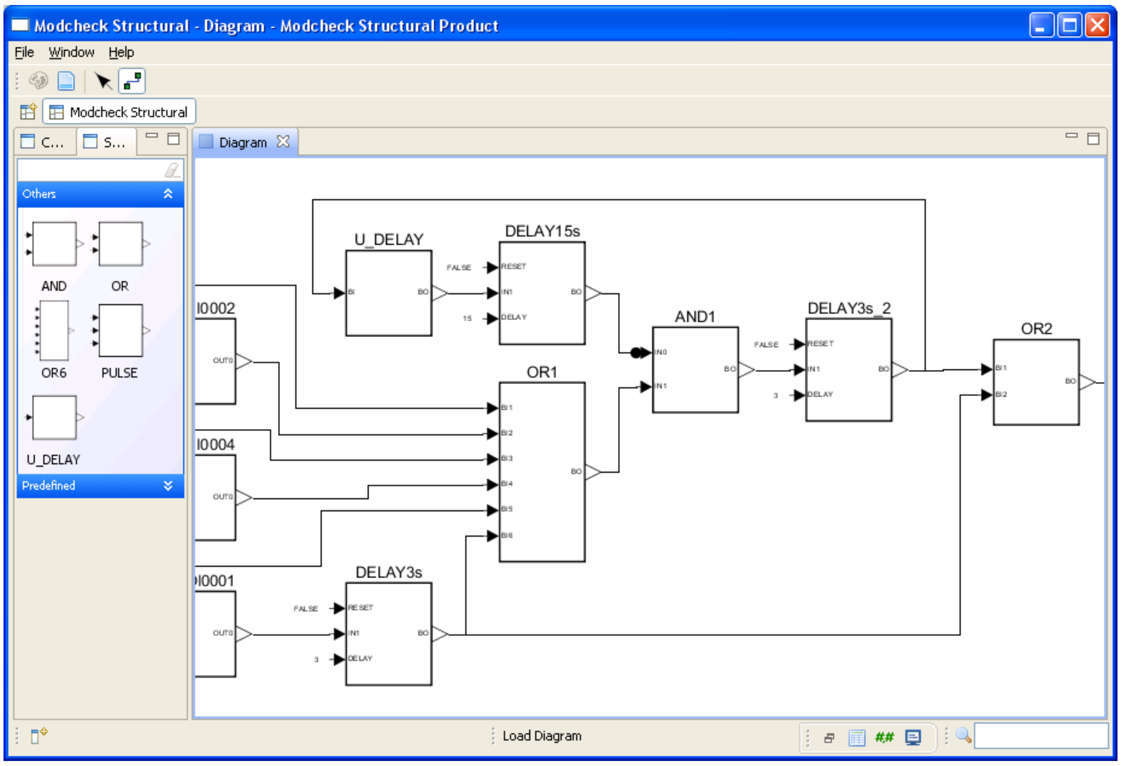


Figure 3. Using the Simantics modelling platform, the system model for verifying function block based designs can be constructed using graphical tools.

3.4 Requirement formalisation for model checking

For most model checkers, the requirements are formalised using temporal logic. Temporal logic is used to describe the changing truth value of different propositions (qualified statements about the state of affairs) over time. In addition to common logical operators, modal operators such as “Globally” (in all states) and “Future” (eventually) are used. The table below lists the modal operators commonly used in the Linear Temporal Logic (LTL), one type of a temporal logic language. In the table, *p* and the *q* stand for propositions, and can either be true or false at any given time (e.g. “temperature > 45”).

Symbolic	Textual	Name	Explanation
○ *p*	**X** *p*	ne**X**t	**X** *p* is true at time *t* if *p* is true at time *t* + *1*.
□ *p*	**G** *p*	**G**lobally	**G** *p* is true at time t if *p* is true at all times *t’* ≥ *t*.
◊ *p*	**F** *p*	**F**inally	**F** *p* is true at time *t* if *p* is true at some time *t’* ≥ *t*.
p **U** *q*	*p* **U** *q*	**U**ntil	*p* **U** *q* is true at time *t* if *q* is true at some time *t’* ≥ *t*, and for all time *t’’* (such that *t* ≤ *t’’* ≤ *t’*) *p* is true.

Table 1, Common modal operators in LTL (modified from [15]).

Even for fairly simple requirements, the temporal logic formulas can become convoluted and difficult to understand. The abstract character of temporal logic and the complexity of the requirement formulas are one of the obstacles limiting the applications of model checking. As a potential solution, the following chapters explore the use of requirement templates to ease the task of requirement formalisation.

4. Ontology-driven requirement templates

We are currently developing a tool concept to help formalisation of requirements. The idea is to store both natural language templates and their corresponding formal presentations in a template library. This allows requirements to be edited in a controlled natural language guided by the ontology and the templates. The formal counterparts of the templates make it possible to automatically generate the requirements for the model checker. So far, we have defined 52 templates on the basis of the property specification patterns introduced in chapter 2.

CNL editor is a Java application that demonstrates syntax and ontology controlled text editing. The purpose is to support designers in writing clear and unambiguous requirement statements with a Controlled Natural Language (CNL). When started, the editor loads the syntax, i.e. the allowed sentence templates, from a file and the vocabulary (ontology) from another file (Figure 4). Then the user can write requirement statements by inserting terms from the ontology to the placeholders in the templates.

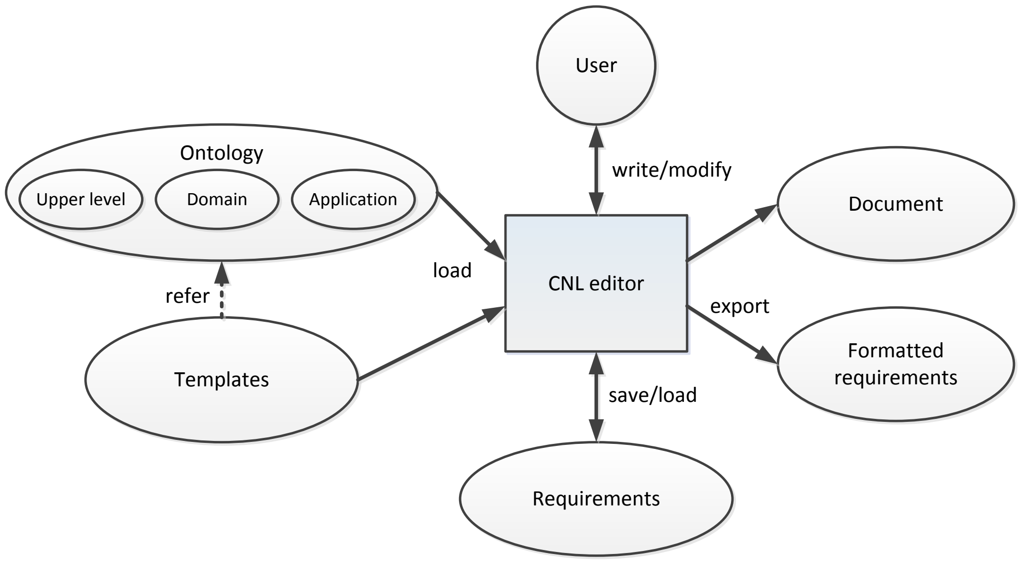


Figure 4. The CNL editor uses the ontology and the template collection to guide the user in editing and formalising the requirements.

Ontologies are currently edited with Protégé in OWL format. The entity classes, such as a "Function" or a "Logical_operator" defined in the ontology are used for writing the templates as illustrated in Figure 5. Templates define allowed natural language sentences by referring to domain concepts and reusable "phrases". The focus is here on formal model checking of control system functions. Therefore, sentences are defined together with corresponding formal representations, in this case the Linear Temporal Logic (LTL). Using these "formulas", the editor is able to export the formalized requirements to a text file for use in a model checker.

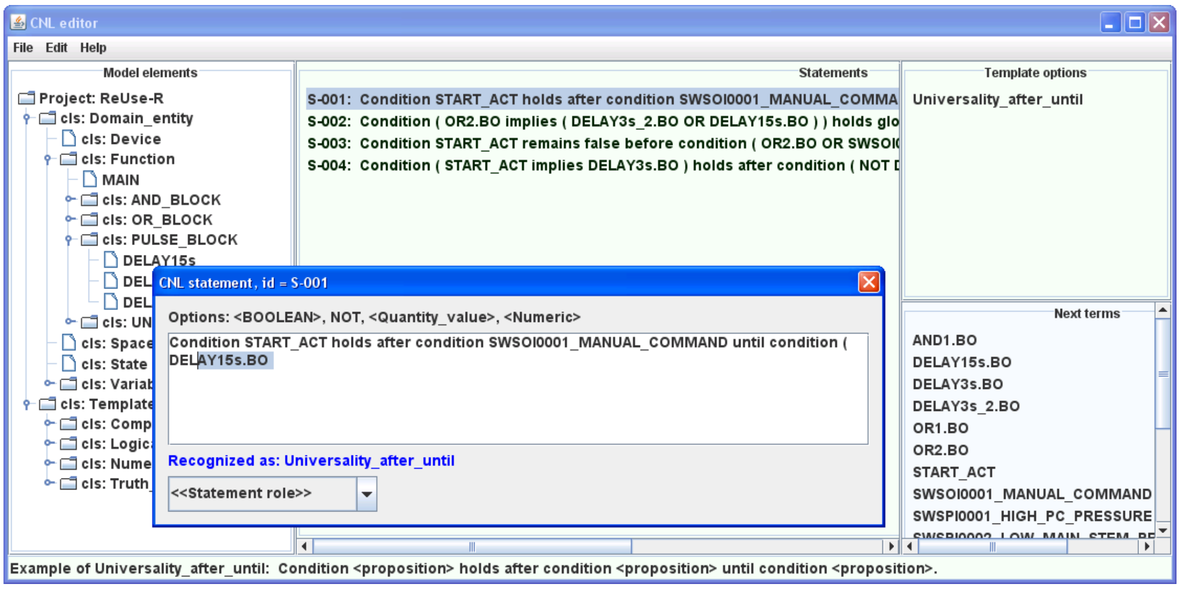
```
phrase _proposition_boolean_value_of
  sentence       <BOOLEAN::BV> of <Function::FV> is <Truth_value::V>
  formula        (<FV>.<BV> = <V>)

phrase _proposition_numeric_comparison
  sentence       <Numeric::VN> is <comparison::CN> <Quantity_value::QN>
  formula        (<VN> <CN> <QN>)

phrase _proposition-combined
  sentence       <_proposition_::PC1> <Logical_operator::OC> <_proposition_::PC2>
  formula        (<PC1> <OC> <PC2>)

template Universality_before
  sentence       Condition <_proposition::P> holds before condition
                 <_proposition::R>.
  formula        LTLSPEC F(<R>) -> (<P> U <R>)
```

Figure 5. Basic idea of phrase and template definition.

5. An example

The example below is based on the Stepwise Shutdown System, a realistic I&C application developed for research purposes. It is a limiting safety system used for gradual movement of a process towards a normal operating state in the case of disturbances. The system is triggered when one of the associated process variables, e.g. reactor temperature, deviates from the limits of normal operation. The functional logic design can be seen in Figure 3 above. For more information, see [16].

The stepwise logic is based on a cycle where the output is first activated for 3 seconds, followed by a 12 seconds idle period. This cycle is repeated as long as the triggering conditions are valid. The requirement specification for the system also states: "The operator shall be able to bypass the 12 seconds waiting period manually at any time. *After the manual activation, the system shall immediately (regardless of the triggering conditions or the cycle phase) send the actuation command for three seconds.*"



CNL editor

File Edit Help

Model elements

Project: ReUse-R
cls: Domain_entity
cls: Device
cls: Function
MAIN
cls: AND_BLOCK
cls: OR_BLOCK
cls: PULSE_BLOCK
DELAY15s
DEL
DEL
cls: UN
cls: Space
cls: State
cls: Variab
cls: Template
cls: Comp
cls: Logic
cls: Nume
cls: Truth

Statements

S-001: Condition START_ACT holds after condition SWSOI0001_MANUAL_COMMA
S-002: Condition (OR2.BO implies (DELAY3s_2.BO OR DELAY15s.BO)) holds glo
S-003: Condition START_ACT remains false before condition (OR2.BO OR SWSOI
S-004: Condition (START_ACT implies DELAY3s.BO) holds after condition (NOT D

Template options

Universality_after_until

CNL statement, id = S-001

Options: <BOOLEAN>, NOT, <Quantity_value>, <Numeric>

Condition START_ACT holds after condition SWSOI0001_MANUAL_COMMAND until condition (DELAY15s.BO

Recognized as: Universality_after_until

<<Statement role>>

Next terms

AND1.BO
DELAY15s.BO
DELAY3s.BO
DELAY3s_2.BO
OR1.BO
OR2.BO
START_ACT
SWSOI0001_MANUAL_COMMAND
SWSPI0001_HIGH_PC_PRESSURE

Example of Universality_after_until: Condition <proposition> holds after condition <proposition> until condition <proposition>.

Figure 6. Writing a requirement in the CNL editor.

We now formalise the latter sentence using the CNL editor. As shown in Figure 6, the tool allows the user to edit a natural language sentence in a text field. When editing, the auto completion feature suggests possible terms from the templates and the ontology. The lists of available templates and next terms are shown on the right side of the main window. Classes and individuals of the ontology can be seen in the tree to the left.

It is probable that the templates differ from the unstructured natural language requirements, such as the one given above. Therefore, the user must rewrite the requirement. To do this, she/he should have prior knowledge of the templates or the opportunity to browse the library for a suitable template. Once knowing what to write, the user can start typing the new requirement. For example, at the point shown in the figure, only one template, "Universality_after_until" matches with the text, but many options are left for the next term. When completed, the requirement statement will read as follows:

> *Condition START_ACT holds after condition SWSOI0001_MANUAL_COMMAND until condition (DELAY3s.clock is equal_to 0).*

The template matching with this sentence is based on the Universality ("P is true") specification pattern found in [7] with the scope "After Q until R". Its LTL formula has the form:

> ***G** (Q **& !**R **-> ((G** P**) | (** P **U** R **))).***

The CNL editor is now able to translate the completed statement into a formula that can be transferred to the NuSMV model checker for verification. When the actual propositions and variables are inserted, the formalized requirement looks like this:

> ***G** (SWSOI0001_MANUAL_COMMAND **& !**((DELAY3s.clock **=** 0)**) -> ((G**(START_ACT)**)|**(START_ACT **U** ((DELAY3s.clock **=** 0)))**)).***

6. Conclusions and future research

Our preliminary results seem promising, at least in the limited domain of model checking and small I&C applications described in terms of functions, ports and variables. It is possible to define a simple ontology and a set of templates that are sufficient to express typical functional requirements. There are, however, several questions that still need to be answered. For example, what kind of domain concepts and templates, and how many of them, would be needed for real-world I&C systems. How should the requirement editor be implemented and integrated with other design and analysis tools? And how should the user interface look like to provide a natural way of working to a requirements engineer or to an analyst wanting to verify the system with a model checker? Our plan is to study these issues further in the Finnish Research Programme on Nuclear Power Plant Safety 2011 – 2014 (SAFIR2014).

7. References

[1] ISO/IEC/IEEE FDIS 29148:2011. Systems and software engineering — Life cycle processes — Requirements engineering.

[2] Mavin, A., Wilkinson, P., Harwood, A. & Novak, M. 2009. EARS (Easy Approach to Requirements Syntax). 17th IEEE International Requirements Engineering Conference.

[3] Hull, E., Jackson, K. & Dick, J. 2002. Requirements engineering. London, Springer-Verlag.

[4] G. Fanmuy, J. Llorens & A. Fraga. Requirements verification in the Industry. Complex Systems Design & Management (CSD&M) 2011, Paris, December 7 – 9

[5] T. Stålhane, S. Farfeleder & O. Daramola. Safety analysis based on requirements. Proc. Enlarged Halden Programme Group Meeting, 2nd – 7th October 2011.

[6] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, “Property specification patterns for finite-state verification”, Proceedings of the second workshop on Formal methods in software practice (FMSP '98). ACM New York, NY, USA, 1998.

[7] H. Alavi, G. Avrunin, J. Corbett, L. Dillon, M. Dwyer, C. Pasareanu, “Specification patterns”, Online at: http://patterns.projects.cis.ksu.edu/

[8] K. Loer, M. Harrison, “Integrating Model Checking with the Industrial Design of Interactive Systems”, 26th International Conference of Software Engineering (ICSE 2004), Edinburgh, Scotland, UK, May 27-28, 2004.

[9] P. T. Monteiro, D. Ropers, R. Mateescu, A. T. Freitas, H. de Jong, “Temporal logic patterns for querying dynamic models of cellular interaction networks”, Bioinformatics, Vol. 24, Issue 16, pp. 227-233. Oxford University Press, 2008.

[10] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, Model Checking, The MITPress, (1999).

[11] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, “Model checking of safety-critical software in the nuclear engineering domain”, Reliability Engineering and System Safety, Vol. 105, pp. 104-113, 2012.

[12] A. Pakonen, T. Mätäsniemi, J. Valkonen, “Model Checking Reveals Hidden Errors in Safety-Critical I&C Software”, 8th International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC & HMIT 2012). San Diego, California, USA, 22 - 26 July 2012. American Nuclear Society, 2012. Pp. 1823–1834, ISBN 978-0-9448-093-0.

[13] T. Karhela, H. Niemistö, A. Villberg, “Open ontology-based integration platform for modeling and simulation in engineering”, International Journal of Modeling, Simulation, and Scientific Computing (IJMSSC). World Scientific. Vol. 3, No. 2, 2012.

[14] “Simantics – a software platform for modelling and simulation”, http://www.simantics.org/ (2012)

[15] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev, ”NuSMV 2.5 User Manual”, available at: http://nusmv.fbk.eu/

[16] J. Valkonen, T. Korvola, L. Lötjönen, J.-E. Holmberg, “Design, implementation and V&V of a Stepwise Shutdown System – Work in progress description of a case study in HARMONICS EU FP7 project”, 37th Enlarged Halden Programme Group Meeting (EHPG 2013), Storefjell, Gol, Norway, 10th-15th March, 2013.