

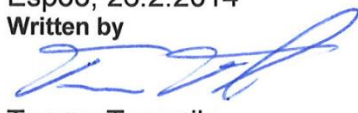
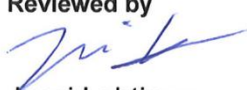
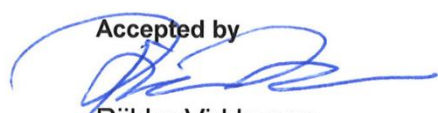


SAREMAN project

Controlled natural language requirements in the design and analysis of safety critical I&C systems

Authors: Teemu Tommila, Antti Pakonen

Confidentiality: Public

Report's title	
Controlled natural language requirements in the design and analysis of safety critical I&C systems	
Customer, contact person, address	Order reference
Nuclear Waste Management Fund P.O. Box 32, FI-00023 GOVERNMENT, Finland	34/2013SAF 12.3.2013
Project name	Project number/Short name
Safety requirements specification and management in nuclear power plants	77380 / SAREMAN-2013
Author(s)	Pages
Teemu Tommila, Antti Pakonen	56 + app. 8 p.
Keywords	Report identification code
Requirements, Controlled Natural Languages, Model checking, Templates, Patterns	VTT-R-01067-14
Summary	
<p>In order to be effectively communicated to all relevant stakeholders, requirements need to be easily understandable, unambiguous, and verifiable. This research report studies the possibilities of textual requirement templates in specifying and analysing safety critical instrumentation and control systems. We use a Controlled Natural Language (CNL) in the shape of a restricted vocabulary and templates for "standardised" requirement statements, where terms of the particular application can be filled-in.</p> <p>While the template-based approach applies to systems engineering in general, our particular interest is in model checking, a formal method to exhaustively show that a system model fulfils its stated functional requirements. Model checking is based on strict requirement formalisation using temporal logic languages, which calls for specific expertise. With natural language templates, oft-used requirement constructs can be re-used without having to work with complex temporal logic expressions.</p> <p>In this report we first discuss the theoretical foundations of system modelling and natural language constructs, and then review related research on methods and tools. On the basis of the literature and VTT's previous experiences in model checking, we suggest a first set of templates intended for model checking function block based control applications and list desired features for a requirement authoring tool based on the templates. Topics for further research include processing of chains of events that are too complex for practical expression in writing, as well as mechanisms for integrating the suggested tool concept into existing model checking, system development, and requirement management tools.</p>	
Confidentiality	Public
Espoo, 26.2.2014	
Written by	Reviewed by
	
Teemu Tommila Senior Scientist	Jussi Lahtinen Research Scientist
	Accepted by
	
	Riikka Virkkunen Head of Research Area
VTT's contact address	
B.O. Box 1000, FI-02044 VTT, Finland	
Distribution (customer and VTT)	
SAFIR2014 Reference group 2	
<p><i>The use of the name of the VTT Technical Research Centre of Finland (VTT) in advertising or publication in part of this report is only permissible with written authorisation from the VTT Technical Research Centre of Finland.</i></p>	

Preface

This report has been written in the research project “Safety requirements specification and management in nuclear power plants” (SAREMAN), a part of the Finnish Research Programme on Nuclear Power Plant Safety 2011–2014 (SAFIR2014). The aim of the SAREMAN project is to develop good practices for requirements definition and management in nuclear power plants. Its focus is on the safety of Instrumentation and Control (I&C) systems. However, due to the multi-disciplinary character of Requirements Engineering (RE), also other plant systems and engineering disciplines are concerned in the project.

We wish to express our gratitude to the representatives of the SAFIR2014 member organisations who have taken the time to engage in discussions and provide valuable input throughout the project.

Espoo 31.1.2014

Authors

Contents

Preface.....	2
Contents.....	3
1. Introduction.....	4
2. The conceptual basis.....	5
2.1 Propositions about states of affairs.....	5
2.2 Elements of the world.....	6
2.3 Artefact functions and actions of goal-oriented agents.....	7
2.4 Temporal and set relationships.....	9
2.5 On the English grammar.....	11
2.6 Towards the anatomy of a natural language proposition.....	12
3. Related research.....	15
3.1 Controlled natural languages.....	15
3.2 Requirement templates.....	17
3.2.1 Requirement Boilerplates.....	18
3.2.2 CESAR Requirement Specification Languages.....	19
3.2.3 Easy Approach to Requirements Syntax (EARS).....	22
3.3 Tools for requirements definition and analysis.....	23
3.3.1 DOORS.....	23
3.3.2 Requirements Authoring Tool (RAT).....	25
3.3.3 Context RDS.....	27
3.3.4 CESAR DODT.....	28
3.3.5 Attempto Controlled English (ACE).....	30
3.3.6 ReqUirements BoileRplate sanlty Checker (RUBRIC).....	31
3.4 Testing automation.....	33
3.5 Model checking and property specification patterns.....	35
3.6 Property Specification Language (PSL).....	38
4. Requirement templates for safety-related I&C systems.....	39
4.1 Vocabulary.....	39
4.2 Sentences.....	42
4.2.1 Propositions of states and events.....	42
4.2.2 Reserved words and requirement scopes.....	43
4.2.3 Types of complete sentences.....	44
5. Working practices and tool support.....	45
5.1 Requirements definition in model checking – a task description.....	45
5.2 Requirements of a template-based requirement editor.....	48
6. Summary and conclusions.....	52
References.....	54
Appendix A: Model checking templates for safety critical I&C.....	57

1. Introduction

The aim of the SAREMAN project is to develop good practices for requirements definition and management in nuclear power plants. Its focus is on the safety of Instrumentation and Control (I&C) systems. However, due to the multi-disciplinary character of Requirements Engineering (RE), also other plant systems and engineering disciplines are concerned in the project.

In order to correctly communicate the needs of various stakeholders, requirements should be unambiguous and easily understandable. Moreover, to cope with the frequent changes and, in particular, to be able to demonstrate the safety of an I&C system to the regulating authority, the project organisation should maintain the traceability among requirements, their sources and the solutions satisfying the requirements.

Even if various diagrams and tabular presentations can be used to express requirements, a big part of requirements are today communicated in written form using natural language. One of the main problems here is that engineers tend use inconsistent terminology, vague words and too complex sentences. Requirements are not verifiable, and they are hard to understand and have several interpretations. Consequently, requirements often fail to do their job in communicating the safety concerns of the regulator and the customer in the supply chain.

These challenges are not limited to requirements. The demand for clarity and traceability concerns all information created in an investment project. The complexity and amount of information emphasises the role of computer tools and electronic information exchange between project participants. Software tools are needed to automate and enhance design activities and to analyse the quality of their results. Even more than in human-human communication, the structure and meaning of the information must be agreed among all participants.

The use of a *Controlled Natural Language* (CNL) is an old and common approach to tackle the challenges of poor written requirements. Standards and textbooks list features of good requirements and give recommendations for well-structured sentences. There are collections of “standardised” requirement statements, i.e. natural language *templates* where terms of the particular application can be filled-in. Some of these templates are also supported by commercial requirements management tools. However, for more advanced software tools of design synthesis and analysis, standardised sentence structures are not enough. The terms inserted into them must come from a limited vocabulary of the domain and the particular application. In other words, the tool must make use of a *domain ontology* and a *model* of the individual I&C system.

This research report studies the possibilities of requirement templates combined with a restricted vocabulary in designing and analysing safety critical I&C systems in nuclear power plants. The goals are two-fold. The first research question concerns the terminology and types of templates needed in I&C applications in general irrespective of computer tools. As already seen in other domains, simple and practical guidance to engineers can rapidly improve the quality of requirements. The lessons learned will be incorporated into the RE guide being developed in the SAREMAN project. The second and primary research question in this report is how templates and ontologies can be used to produce requirements that are sufficiently formal to be processed by computer tools. One issue here is how the I&C-oriented templates could be applied in existing requirements management tools. Our particular interest is, however, in *model checking*. In this methodology, a formal model of both the system and the requirements is needed to exhaustively check whether the requirements are satisfied by the suggested I&C functions. We expect that templates help us formalise human-readable requirements to the temporal logic presentation used in model checker tools. The results of this study can hopefully be integrated to the model checking tools studied in related research activities, especially in the SARANA project.

The rest of this report is organised as follows. Chapter 2 discusses the theoretical foundations of the domain ontology and natural language statements. This discussion makes use of our previous work on conceptual modelling and role of requirements in systems engineering (SAREMAN 2013a). Recognising that templates are already widely used in engineering design, chapter 3 presents some examples of recommendations and tools particularly relevant to us. As a combination of these ingredients, we suggest in chapter 4 a first set of templates particularly intended for model checking function block based control applications. More detailed descriptions of the templates are given in appendix A. Next, we discuss in chapter 5 the desired features of a requirement editor tool in the context of model checking. Finally, chapter 6 summarises the findings of the study and gives some suggestions for next steps.

2. The conceptual basis

In SAREMAN, we speak about model-based engineering (SAREMAN 2013a). The *system model* describes a future world in terms of various *model elements*. The main purpose of the model is to communicate ideas. Therefore, model elements can be understood as *statements* of stakeholders and designers. Adapted from human communication and the speech-act theory, statements have a *communicative role* indicating whether they should be interpreted as facts, requirements, intentions, etc. Some of the model elements represent entities that will eventually exist in the real world (e.g. devices and software). Some others are abstractions that rather describe what will happen (occur) during system operation (function, event). Some model elements are statements about properties (e.g. accuracy) and relationships (location of a device) between entities. In model checking the focus is on functional requirements but in the wider context of systems engineering all kinds of statements should be considered. Below we discuss informally a few obvious ingredients of our natural language templates.

2.1 Propositions about states of affairs

We adopt here the term “state of affairs” from philosophy. A *state of affairs* more or less corresponds to a “situation”. It is the way the actual world must be in order to make a given *proposition* (statement) about the actual world true. Propositions can describe also attitudes like beliefs and desires. For example, the statement “Socrates is wise” is said to express the opinion that Socrates is wise and also to describe the actual state of affairs of Socrates being wise (<http://plato.stanford.edu/entries/states-of-affairs/>). The *communicative role* of a statement just above indicates the correct interpretation. Moreover, a state of affairs is a possible situation in the world. So, a situation can have an associated probability. A proposition about that situation can be true or false or perhaps have a confidence level between 0 and 1.

We are speaking about dynamic systems where situations change over time. Accordingly, state of affairs refers to a past, current or future situation out there in the real world. Of course, we need also logical combinations of these basic propositions, as well as the events of propositions turning from false to true and vice versa. So, a state of affairs can consist of a set of consecutive and simultaneous (past, current and future) situations and events (Figure 1). Given that our systems are complex, it makes sense to think of complex states of affairs consisting of simpler ones, both in terms of time and decomposition of the world to subsystems. For example, “system A is in state X and then in state Y” or “system A is in state X and system B in state Y”.

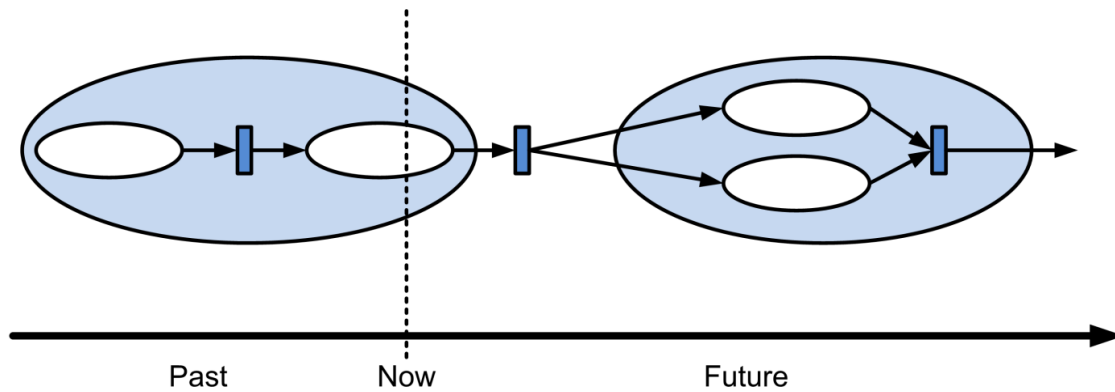


Figure 1. Evolution of states of affairs.

In engineering design, a statement is a (simple or complex) proposition about a future state of affairs. In our case, the *communicative role* is usually “requirement” but, as claimed in the SAREMAN conceptual model, it should not be embedded into the proposition but presented as a separate attribute. In other words, we are looking for templates that are applicable not only to requirements but also to facts, assumptions, etc. Moreover, we are speaking about statements that can be expressed textually in a controlled natural language as one sentence. For more complex chains and combinations of states of affairs we would need longer stories or graphical presentations like sequence diagrams. Such techniques are out of our scope at this point.

2.2 Elements of the world

So, we are trying to collect templates for propositions about a future world (when speaking about design) or an existing one (when testing is considered). To do this, it is necessary to first imagine what kind of entities and phenomena we might encounter in that world. These entities are discussed in the SAREMAN conceptual model (SAREMAN 2013a). Figure 2 illustrates the main concepts. A nuclear power plant is a physical, real-world *system* that can be hierarchically decomposed into parts here called *NPP elements*. The human organisation, various technical systems and the operational environment are the main components. People and devices are located in *spaces* (e.g. rooms) that are bounded by *structures* like walls. Technical systems (process systems and I&C systems) consist of devices, software and structural elements (e.g. cabinets). NPP elements can be in various *operational states* (e.g. idle or running) and, depending on their state, carry out certain *functions*. For example, a control system can have the capability of controlling the feed water flow to the reactor. As suggested in (SAREMAN 2013a), we reserve the term *system function* for model elements used to by designer to specify the intended capabilities of the system. *Purpose* in turn links the system and its function to the activity it is used for, either as intended by the designer or actually by the user in a specific situation.

The functions provided by NPP elements are combined to carry out intended *activities*, such as energy production. Activities can be decomposed into subactivities, concrete *tasks* (e.g. plant shutdown) and finally to atomic *actions*. When actually carried out, activities are allocated to suitable NPP elements. NPP elements perform actions on some other NPP elements by using operations (actions) provided by their functions. For example, to start controlling the feed water flow the operator turns the flow controller into the auto mode. Note that atomic actions can be considered as a subtype of *event*. Physical system elements and their functions represent a static view to the system but are, in principle, capable of generating all possible *behaviours* of the system, i.e. occurrences that might be observed in the real world.

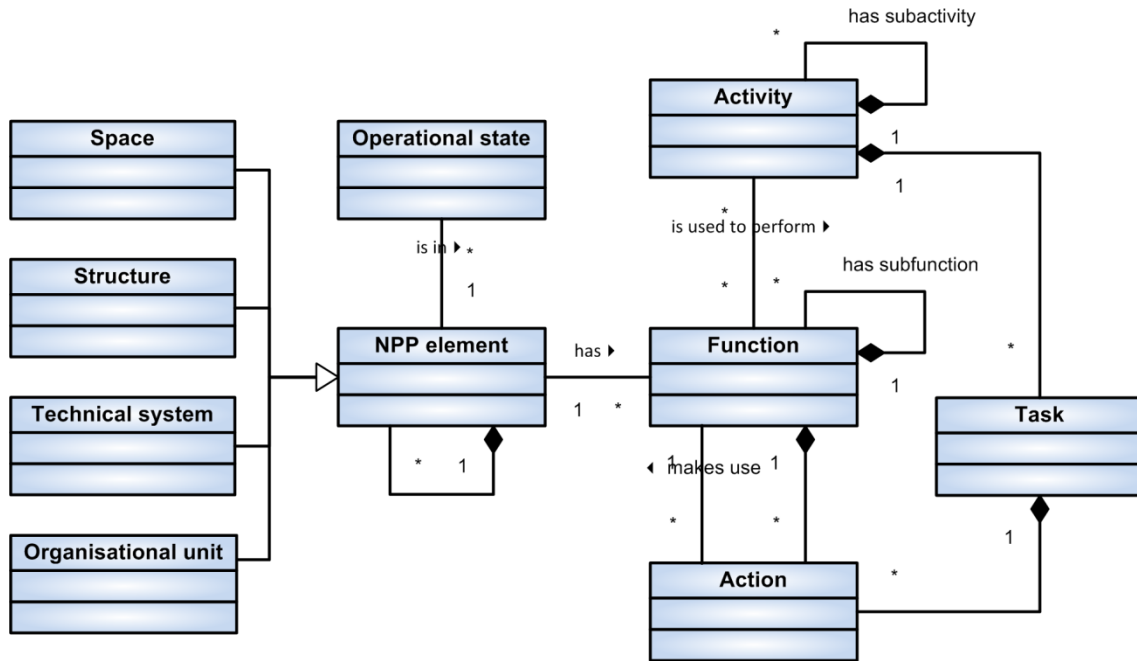


Figure 2. Main entities in a power plant model.

All entities can have a (possibly large) set of properties of various types. Some of them are fixed (e.g. size), some configurable (tuning parameters of a PID controller) and variable (reactor pressure). Moreover, entities are related to each other in many ways. Also these relationships can have properties.

During design time, the real NPP elements may not yet have been specified. For example, we know that there will be an I&C system that has a function, but have no idea of the subsystem or piece of software that will implement that function. So, when speaking about the system we must make statements about abstract functions that exist in the system model but never in the real world. It should be also remembered that the design organisation can be considered as a system of its own. Therefore, states of affairs and propositions can refer to situations in the development project of a technical system.

With these concepts we can now return to propositions about possible states of affairs and try to outline a classification of the types of propositions needed to describe a nuclear power plant and its I&C systems. Here are some examples:

- NPP element exists (as part of a larger entity)
- NPP element has a function
- property of an entity (or all entities in a set) has a value
- property value of an entity is higher than the property of another entity
- NPP element is in an operational state
- NPP element is located in a space
- NPP element performs an action
- NPP element performs and activity or participates in an activity

2.3 Artefact functions and actions of goal-oriented agents

An important group of requirements, i.e. *functional requirements*, describe, in writing or graphical means, the behaviour of a system, for example as its reactions to external stimuli or its ability to perform certain activities. We claim that a consistent taxonomy of activities,

actions and functions would guide designers in writing meaningful and unambiguous requirement statements. Moreover, such a taxonomy would be needed as a basis of computer tools.

In his extensive review, Crilly (2010) states that despite the importance of function to design and use, there is no stable or generally accepted definition of function. Outside the design literature, many definitions and classifications can be found, e.g., in philosophy, sociology and art theory. In engineering design, the focus has been on transforming inputs to outputs. However, this leaves out many non-transformative functions, such as guiding and supporting. It should be also noted that some functions are non-technical, i.e. serving aesthetic and social purposes. Finally, a distinction can be made between system functions as intended by the designer and those experienced by the user. In essence, an artefact is assigned a function if it is taken to have the capacity to play some role for an agent using the artefact in some context. (Crilly 2010)

It is clear that a complete taxonomy of activities, actions and functions is beyond our resources and our focus on control systems and model checking. However, because of our more general interest in controlled natural languages and systems engineering we summarise below a few sources that might be used as a starting point.

We observe that the real world consists of various man-made artefacts. They have a *purpose*, either defined by the designer in advance (intended) or by the user in a specific situation (actual). Some artefacts are passive (e.g. structures) and some reactive (protection system) while some others can be considered as proactive and goal-oriented agents (people and some computer-based systems). In addition to designed behaviours, natural phenomena need to be considered in system models.

A theoretical framework of acting based on the work of von Wright is discussed by Morten Lind (2005). Goal-oriented agents can oppose or promote certain states of affairs, either by active interventions or by doing nothing and letting things happen. This leads to a set of elementary types of action as illustrated in Figure 3.

		Promotive		Opposive	
Interventions		produce p	produce ~p	destroy p	destroy ~p
		maintain p	maintain ~p	suppress p	suppress ~p
Omissions		let p happen	let ~p happen	let p disappear	let ~p disappear
		let p remain	let ~p remain	let p remain absent	let p remain absent

Figure 3. Interpretations of the elementary types of acting (Lind 2005).

Interestingly, similar types of functions can be found in I&C systems. For example, the purpose of regulatory control functions (PID loops) is typically to maintain a stable state while interlocks prevent unwanted situations from occurring. Protection functions in turn actively produce a safe state thereby destroying the hazardous situation. Omissions remind us of “negative requirements”. I&C systems shall not perform spurious activations or interfere with the functions of other systems or the user.

So, the elementary types above might be useful as a top-level classification. Additional and more practical taxonomies can be found from publications and standards in the engineering domain. While criticised for ontological inconsistencies and focus on flows only (Garbacz 2006), the classifications by Hirtz et al. (2002) represent a synthesis of a long research tradition in engineering design. Verbs like connect, convert, support, signal and control magnitude are examples from the function taxonomy. Similar terms can be found in IEC 81346-2 (2009) concerning structuring principles and reference designations of industrial systems (Figure 4). These kinds of guidelines and standards could be used as basis of a controlled vocabulary and taxonomy of verbs needed for describing I&C functions.

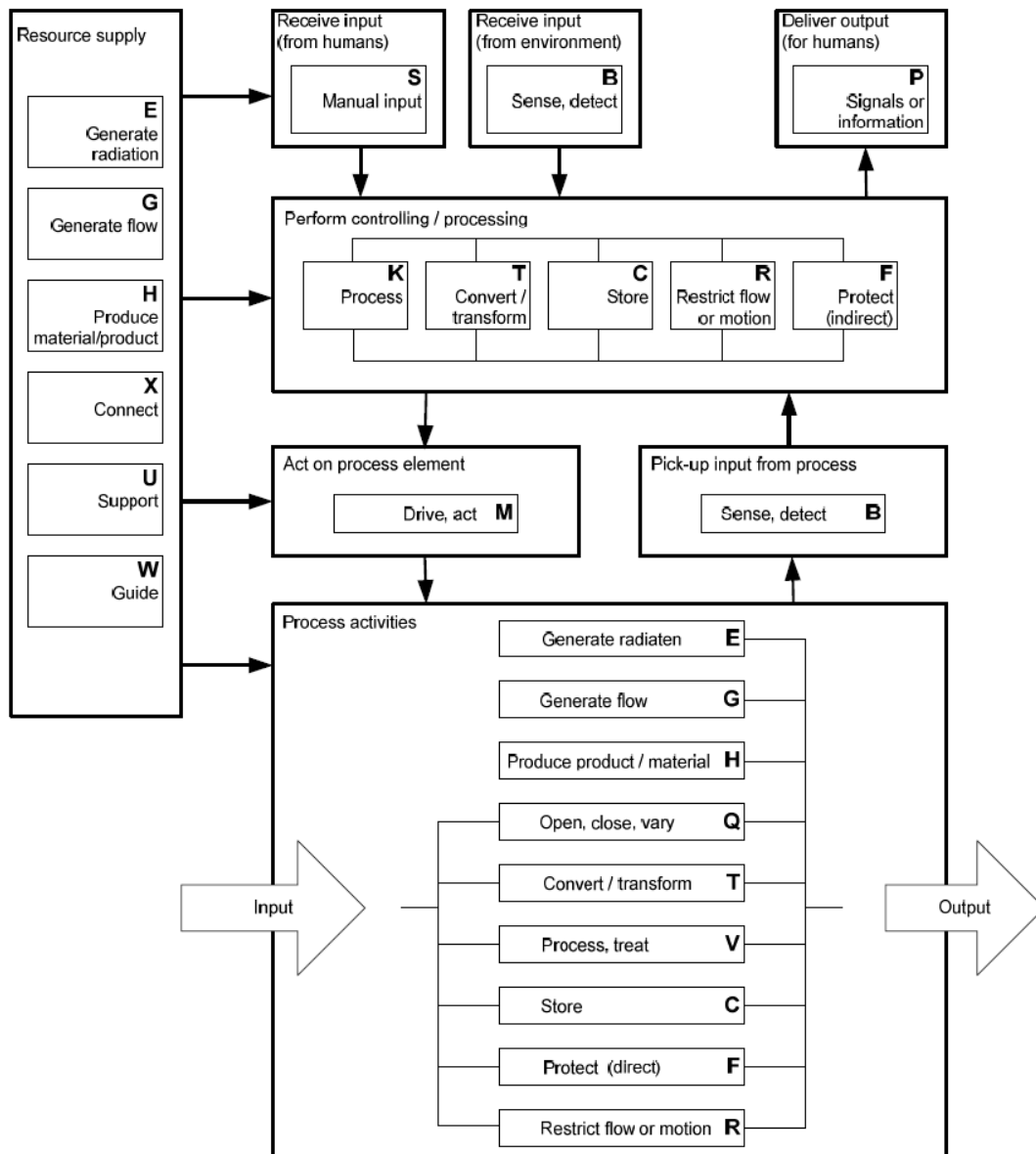


Figure 4. Classes of activities according to intended purpose or task (IEC 81346-2 2009)

2.4 Temporal and set relationships

Thinking about dynamic systems also leads us to complex states of affairs consisting of several sequential and concurrent situations. Therefore, we need to classify relationships between temporal intervals and actions/events. Events can be understood as points of time or very small intervals that start or end a state of affairs, i.e. when the corresponding proposition is made or otherwise becomes true or false. Here is one list suggested by Allen (1984):

- DURING (t1, t2): Time interval t1 is fully contained within t2;
- STARTS(t1, t2): Time interval t1 shares the same beginning as t2, but ends before t2 ends;
- FINISHES(t1, t2): Time interval t1 shares the same end as t2, but begins after t2 begins;
- BEFORE(t1, t2): Time interval t1 is before interval t2, and they do not overlap in any way;
- OVERLAP(t1, t2): Interval t1 starts before t2, and they overlap;
- MEETS(t1, t2): Interval t1 is before interval t2, but there is no interval between them, i.e., t1 ends where t2 starts;
- EQUAL(t1, t2): t1 and t2 are the same interval.

Including the time dimension to states of affairs leads to two types of complex propositions: 1) states of affairs and events occur in a defined order determined by the temporal relationships above; and 2) specific propositions hold during a temporal interval. Note that many kinds of natural language expressions can be constructed by using everyday terms and changing the order of intervals. Note also that the relationships do not necessarily imply causal dependencies between the intervals. However, it would be meaningless to speak about random coincidences in the context of design and testing. Letting E be an event and P and Q propositions corresponding to two time periods might lead us to the following examples:

- DURING(E, P): E occurs while P holds. (Requirement or design intent, P enables E)
- STARTS(E, P): When E happens, P becomes immediately true. (Expected behaviour of a system)
- FINISHES(E, P): When E happens, P becomes immediately false. (Expected behaviour of a system)
- BEFORE(E, P): When E happens, P becomes true after x time units.
- BEFORE(P, E): E occurs after P has turned to false.
- BEFORE(P, Q) OR BEFORE(Q, P): P and Q shall not be true simultaneously (Expected behaviour of a system)
- OVERLAP(P, Q): Q becomes true before P turns to false. (Expected behaviour of a system)
- MEETS(P, Q): When P turns to false, Q becomes immediately true. (Expected behaviour of a system)
- EQUAL(P, Q): P and Q shall be simultaneously true. (Requirement)

We can also think of time periods as sets of time points. So, some of the temporal relationships correspond to relationships between sets, for example:

- EQUAL(A, B): A and B have exactly the same elements (corresponds EQUAL time periods)
- SUBSET(A, B): Every element of A is in B (DURING)

- PROPER SUBSET(A, B): Every element of A is in B except for at least one (STARTS, FINISHES)
- DISJOINT(A, B): A and B have no elements in common (no OVERLAP)
- COMPLETE($A_1 \dots A_n$, B): The union of $A_1 \dots A_n$ is EQUAL to B.

When expressed in a suitable way, these relationships can be useful in natural language statements. For example, a requirement might hold for every field instrument of a given type. Or, the measurement ranges of two instruments cover the whole range of a process variable, including also abnormal and accident conditions.

2.5 On the English grammar

As discussed in section 2.1, natural language sentences are statements about simple or complex states of affairs. So, the structure of a natural language, in this case English, should be considered as one starting point. In written English *sentences* consist of one (simple sentence) or more (compound sentence) *clauses*. The clauses in a compound sentence are joined by co-ordinating conjunctions (and, but, so). A complex sentence has a *main clause* and one or more subordinate *clauses*. Clauses are made of *phrases*, i.e. groups of one or more grammatically linked words that do not have subject and predicate. Clauses in English have at least two parts: a noun phrase and a verb phrase. Types of phrases include (<http://grammar.about.com>, <http://learnenglish.britishcouncil.org/english-grammar/clause-phrase-and-sentence>):

- Adjective phrase modifies a noun. It may appear before the noun (sweet coconut) or after a linking verb (shampoo tastes funny). Adjectives (big) can be comparative (bigger) and superlative adjectives, intensifiers (very) and mitigators (fairly), and noun modifiers (metal box). A lot of adjectives are made from verbs by adding -ing or -ed (amusing).
- Adverb phrase modifies a verb, an adjective, or another adverb. Adverbials are used to say how, where, when or how certainly something is done. There are several types adverbs and adverbial phrases: contrast, reason, place (near the wall, everywhere), purpose, result, time (frequently), condition, manner (slowly).
- Noun phrase usually functions as a subject or object in a clause. The simplest noun phrase consists of a single noun possibly accompanied by modifiers or determiners (such as the, a, her) or a pronoun. Sometimes the noun phrase begins with a quantifier (all, some).
- Prepositional phrase is a combination of a preposition (above, before, within) and a noun phrase. Prepositions convey the various relationships: agency (by); comparison (like, as); direction (to, toward, through); place (at, by, on); possession (of); purpose (for); source (from, out of); and time (at, before, on)
- Verb phrase includes a main verb and its auxiliaries (such as have, do, or will) that determine the mood (attitude), modality (necessity, uncertainty, ability, or permission, e.g. can, could, may, might, must, ought, shall, should), tense (past, present, future), or aspect (completion, duration, or repetition) of another verb. The structure of the verb clause depends on the type of the verb (transitive, intransitive, etc.)

Even if a natural language is rich in forms and nuances, it can also express the fundamental logical aspects that we need. We should just carefully select a subset of words and sentence structures that are correctly understood by most readers. Remembering the recommendations for well-formed requirements (SAREMAN 2013b), it is clear that all common words and phrases, e.g. vague words and ambiguous pronoun references, can NOT be used in our templates.

2.6 Towards the anatomy of a natural language proposition

To summarise the philosophical considerations above, we try to identify a small set of features that could be used to structure practical statements and to define potential templates.

General guidelines for well-formed requirements can be found in standards and textbooks. A summary is given in the SAREMAN RE guide (SAREMAN 2013b). For example, passive voice, weak words and error-prone sentence structures should be avoided since they lead to difficulties in design verification and human communication. The decision depends, however, on the domain and expected users. Sometimes we might want to express soft goals in addition to strict requirements. In particular, the exact interpretation of repeatedly used *keywords*, such as when, while, then, if, after, all, etc., needs to be defined. The table below gives some suggestions.

Table 1. Interpretation of repeatedly used keywords.

Keyword	Recommended usage
While	Period of time, e.g. “ While the plant is in the shut-down state, ...”, or “ While the field operator is working inside the containment,”.
When	Event or action, e.g. “ When the operator pushes a button, ...”.
Whenever	Same as “when” with the idea that there may be many occurrences or the event?
Where	Conditional set of entities irrespective of time, e.g. “ Where a motor-operated valve is installed inside the containment, it shall” (see Mavin et al. 2009, 2010).
If	a) Statement of a logical implication (causality) irrespective of time, e.g. “ If the pressure is higher than x, the vessel may burst”. b) Decision making in the course of an action, e.g. “While performing a periodic check, if the filter is dirty, the technician shall replace it”.
Then	a) Consequence of a logical implication, e.g. “If, then”. b) Temporal order of events and states, e.g. “....and then ...”.

As suggested in ISO 29148 (2011), a statement can be divided into three main parts: condition, main body and constraint. The main body contains the key proposition, simple or complex. The condition (scope) says under what circumstances the main body holds, and the constraint refines the body, for example in terms of performance and time. Table 2 below gives some examples.

Table 2. Requirement statements divided into condition, main body, and constraint.

Condition	Main body	Constraint	Comments
-	System has two redundant temperature sensors.	-	Existence of the sensors.
-	The maximum operating pressure of the reactor vessel is 0.5 MPa.	-	Property of a device.
-	The process controller is located in the cross connection room.		Relationship in system structure

Condition	Main body	Constraint	Comments
-	The two redundant temperature sensors are powered by different power supplies.	-	Complex relationship.
While system is in AUTO mode,	when temperature rises above 100 C, system opens the relief valve	within 10 seconds to prevent over pressure.	State and event triggered behaviour with a time limit. System as the actor, open as type of acting. Rationale included.
Where a sensor is used for a safety-relate function,	the sensor must have a redundant power supply.		

Table 3 below lists types of clauses with some examples. The terms in angle brackets, e.g. <entity>, depend on the application domain and can't be combined in arbitrary ways. For example, it is obviously nonsense to say that "a <function> is located in a <room>". In addition to domain entities and properties, we need many common words like verbs and adverbs in the recommended sentence structures. They should provide some freedom to the author but still convey a unique meaning.

Complete sentences can be built by combining standard clauses with logical (and, or...) and temporal (after, then...) conjunctions. In the general case, the number of terms and sentence structures needed in systems engineering is, of course, enormous. Therefore, we will focus on a very limited domain in chapter 4.

Table 3. Examples of requirement clause structures.

Clause type	Examples of sentence structures
System structure (usually static)	
Composition	<entity> has <entity entity list> [as its part] <entity> is part of <entity>
Arrangement	<entity> is located in <entity> <entity> is located near to <entity> distance between <entity> and <entity> is [<comparison>] <value>
System state	
Property	<property> of <entity> is <value range> <unit of measurement> <property> of <entity> is [<comparison>] <value> <unit of measurement> <property> of <entity> is [<comparison>] <property> of <entity>
Variable value	<variable> is <value range> <variable> is <comparison> <value> <variable> is <comparison> <variable>
Operational state	<entity> is in <operational state> state
Operating mode	<entity> is in <operating mode> mode
Event (duration = 0)	

Clause type	Examples of sentence structures
Composition change	<entity> is added to <entity> <entity> is removed <entity>
Arrangement change	<entity> enters into <entity> <entity> comes near to <entity> distance between <entity> and <entity> becomes [<comparison>] <value> <work item> is received
Property change	<property> of <entity> becomes <comparison> <value> <property> of <entity> becomes <comparison> <property> of <entity>
Variable value change	<variable> becomes <comparison> <value> <variable> becomes <comparison> <variable>
Operational state change	<entity> enters <operational state> state
Operating mode change	<entity> enters <operating mode> mode
Action	<entity> performs <action> <entity> starts <activity> <entity> <verb> <entity> <entity> sets <property> of <entity> <entity> sets <variable> to a value that is <comparison> <value> <entity> sets <entity> to <operational state> state
Activity (duration > 0)	
Maintaining	<entity> keeps <property> of <entity> in the range <range>
Producing	<entity> brings <entity> into <operational state> state by ... <entity> makes <entity> perform <action>
Preventing	<entity> prevents the value of <variable> from becoming <comparison> <value> <entity> prevents <entity> from performing <action>
Allowing (i.e. not preventing)	<entity> allows the <property> of <entity> to become <comparison> <value> <entity> allows <entity> to <verb> <entity> <entity> allows the <property> of <entity> remain in <value>
Supporting	<entity> helps <entity> to <verb> <entity>
Performing	<entity> performs <activity>
Timing & performance	
While	while <system state activity>
Within	<entity> performs <action> within <duration>
Period	<entity> performs <activity> for a period of <duration>
Before	<entity> enters <operational state> state before <event> <entity> enters <operational state> state within <duration>
After	<entity> performs <action> after <event duration>

Clause type	Examples of sentence structures
Until	<entity> performs <activity> until <event state>
Always	<property> of <entity> is always <value>
Never	<entity> never performs <action>
When	when <event> occurs, <entity> performs <action>
Order/Then	when <event_1> occurs and then <event_2> occurs, <entity> performs <action>
Performance	<entity> performs <action> with the rate of <value> per <time unit> <entity> never performs <action> with the accuracy of <value> <unit of measurement>
Sets	
Where	where <property> of <entity type> is [<comparison>] <value>, ... where <entity type> has <entity type> as its part, ...
Misc.	
Possibility	it is [not] possible that
Probability	<event> occurs with probability of <value>
Decision	if <property> of <entity type> is [<comparison>] <value>, ...

3. Related research

The idea of using predefined templates for requirements authoring is not new but can be found in textbooks, standards and many design tools, in particular in the areas of systems engineering and software development. Consequently, there are lessons to be learned for control system design and model checking. This chapter describes some of the most relevant approaches.

3.1 Controlled natural languages

Natural Language Processing (NLP) is a field of computer science, artificial intelligence and linguistics concerned with the interactions between computers and human languages in the form of text or speech. NLP includes many areas from semantic analysis and machine translation to word stemming. Examples of tasks needed in NLP are Part-Of-Speech (POS) tagging, stemming and parsing. POS tagging determines the linguistic category of the words, e.g. nouns and verbs. Stemming finds the root (stem) of a word for inflected forms, e.g., the singular for a plural word. Parsing performs the grammatical analysis of a sentence resulting in a syntax tree. NLP relies on formal models of language at the levels of phonology and phonetics, morphology, syntax, semantics and discourse. These formal models include, for example, state machines, rule systems, logic and probabilistic models. There are many natural language processing toolkits available for NLP. Also the technical committee TC 37 within the International Organization for Standardization (ISO) prepares standards concerning methodology and principles for terminology and language resources. (Several sources, e.g. Farfeleder et al. 2011, <http://www.cs.colorado.edu/~martin/SLP/Updates/1.pdf>, Wikipedia)

In this report, we are most interested in a specific area of NLP, namely *Controlled Natural Languages* (CNL). As defined by Kuhn (2013), a controlled natural language is a constructed (written) language that is based on a natural language, being more restrictive concerning lexicon, syntax and/or semantics while preserving most of its natural properties. Especially

during the last four decades, a wide variety of such languages have been designed, most of them based on English. Usually, controlled natural languages fall into two major types (both relevant to us): those that improve readability for human readers, and those that enable automatic semantic analysis. The first type is used in the industry to increase the quality of technical documentation. The second type has a formal logical basis. (Kuhn 2013, Fuchs 2008, <https://sites.google.com/site/controllednaturallanguage/>)

Kuhn (2013) has reviewed a large number of controlled natural languages, and we pick some examples from his report. ASD-STE and SLANG are examples of industrial applications of CNL. The *ASD Simplified Technical English* has been developed for the aerospace industry to make texts easier to understand, especially for non-native speakers. The *Standard Language* (SLANG) is a language developed at Ford Motor Company for writing instructions for component and vehicle assembly. Based on these instructions, the system can automatically generate a list of required elements and calculate labour times.

A third example of industrial CNL is the *Gellish English* designed to be a common data language for industry. Basically, its underlying data model consists of simple subject–predicate–object triplets in the form of fixed phrases such as “A is a specialization of B”. Gellish builds upon a fixed upper ontology with a large number of predefined concepts and relation types. Texts in Gellish can be transformed into a formal tabular representation. The semantics of the language is not fully formalized. (Kuhn 2013)

The Gellish data model and Gellish English (van Renssen 2005, <http://www.gellish.net>) are based on various international standards, such as ISO 10303 (STEP), ISO 15926, as well as terminology sources like ISO 16354 and IEC 60050. Its dictionary contains several domain taxonomies, for example an upper ontology of concepts and relation types, units of measure, activities, events and functions, as well as physical objects, such as process units, buildings, electrical and instrumentation equipment, materials of constructions and organizations. The triple structures can be extended with auxiliary facts, e.g. with a textual requirement or an indication of the interpretation of the expression as a statement, a question, an answer, a denial, etc¹. In fact, the Gellish data model can also be used to express requirements in an unambiguous and computer interpretable way (Figure 5).

As a further example, Attempto Controlled English (ACE) is a CNL with an automatic and unambiguous translation into first-order logic. The most notable features of ACE include complex noun phrases, plurals, anaphoric references, subordinated clauses, modality, and questions. ACE will be described in more detail in section 3.3.5.

Finally, the Common Logic Controlled English (CLCE) is a language that can be translated into first-order logic. It is related to the ISO standard 24707 for Common Logic that is a semantic foundation for an open-ended family of languages, such as predicate calculus, conceptual graphs, and CLCE. Some of the most important syntax restrictions are: no plural nouns, only present tense, and variables instead of pronouns. The wish list of Sowa (2011) includes a way to write statements about propositions, e.g. to express uncertainty and various types of modality (necessity and possibility; knowledge and belief; obligation and permission).

¹ Derived from the Speech Act theory that has also influenced the SAREMAN conceptual model.

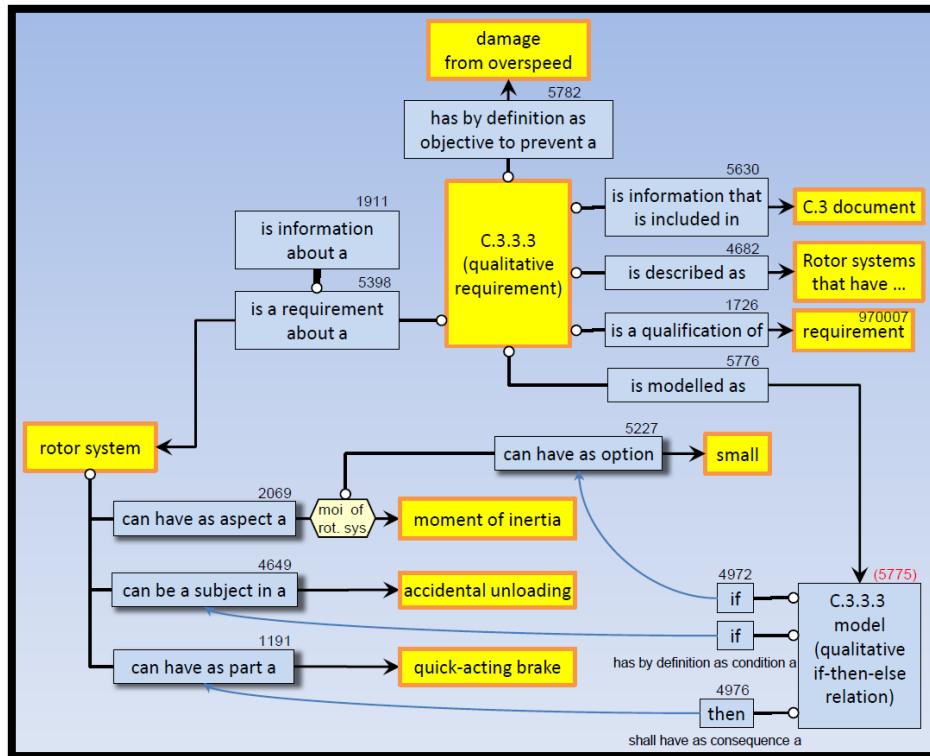


Figure 5. Model of the requirement “C.3.3.3 Rotor systems that have low inertia and are subject to accidental unloading should be equipped with a quick-acting brake to prevent damage from overspeed.” (van Renssen, *From multiple standards documents to a single requirements model*, slides)

3.2 Requirement templates

We use often the term “template”. So, it must be defined in some way. Basically a *template* is understood here as a standardized and pre-formatted partial solution that, when combined with case-specific data, can be used to produce new artefacts, for example documents or source code in the Java language. Usually the application-specific data is just added to the variation points, placeholders, of the template. A *template processor* is a more complex software tool designed to combine one or more templates with the data (e.g. parameters) to produce new artefacts. It typically includes features of programming languages, such as variables, functions, conditional evaluation and loops. (see en.wikipedia.org/wiki/Template_processor)

Templates are one form of reusable design knowledge. For example, software *components* and design *patterns* have the same purpose but a different meaning. Components, such as function blocks (types) in a PLC application programming environment, can be instantiated, parameterised and “wired” together to form a control application. Patterns are usually applied and combined in more creative ways. Patterns were first applied in architecture, but became popular also in software engineering, when the “Gang of Four” published their book on reusable design patterns (Gamma et al. 1994). A pattern describes a recurring problem in a design context and presents a well-proven solution for the problem (Buschmann, Henney & Schmidt 2007). A pattern also tells when it is applicable and describes the main issues to be considered when it is used. An organised collection of patterns with an associated development process is called a *pattern language*. Note that in model checking literature (section 3.5) the term “pattern” is also used for reusable property specifications. Design pattern is, however, a different concept. To avoid confusion, we try here to always speak about templates rather than patterns.

Numerous templates for natural language requirements have been proposed over the years (Arora et al. 2013a). Below we describe some of them.

3.2.1 Requirement Boilerplates

The term related to template, “boilerplate” was originally used to identify the builder of a steam boiler. Today it often refers to text that can be reused without being changed much from the original. Many computer programmers often use the term boilerplate code. In contractual law, the term “boilerplate language” describes the parts of a contract that are considered standard. (from <http://en.wikipedia.org/wiki/Boilerplate>)

Table 4. Examples of Boilerplate clauses (www.requirementsengineering.info).

ID	BOILERPLATE CLAUSES	CLASSIFICATION & COMPLETE EXAMPLE
BP2	The <system function> shall able to <action> <entity>	Function The <i>missile launcher</i> shall able to <i>fire missiles</i>
BP35	... at least <quantity> times per <time unit>	Function/Rapidity (Maximise, Exceed) The <i>missile launcher</i> shall able to <i>fire missiles</i> at least <i>15 times per minute</i>
BP36	... while <operational condition>	Function/Rapidity/Mode The <i>missile launcher</i> shall able to <i>fire missiles</i> at least <i>15 times per minute</i> while <i>in winds upto Beaufort scale 8</i>
BP42	... within <quantity> <time unit>s from <event>	Function/Timeliness (Minimise, Do not exceed) The <i>missile launcher</i> shall able to <i>fire missiles</i> within <i>5 seconds</i> from <i>receipt of warning signal</i>
BP43	... while <operational condition>	Function/Timeliness/Mode The <i>missile launcher</i> shall able to <i>fire missiles</i> within <i>5 seconds</i> from <i>receipt of warning signal</i> while <i>in winds upto Beaufort scale 8</i>
BP12	... for a sustained period of at least <quantity> <time unit>	Function/Sustainability (Maximise, Exceed) The <i>missile launcher</i> shall able to <i>fire missiles</i> for a sustained period of at least <i>20 minutes</i>

Hull, Jackson and Dick (2002) first used the term boilerplate to refer to a textual requirement template. A boilerplate like “<system>shall<action>” consists of a sequence of attributes and fixed syntax elements. During instantiation, textual values are assigned to the attributes of the boilerplates. The basic building block of a boilerplate is a boilerplate clause classified, e.g., as capability, function, timeliness, etc. In addition, each clause may have a goal type indicating the general objective being articulated, e.g. minimise something, maximise something, exceed some value, etc. These keywords can help considerably in processing and organising the requirements. Complex requirements can be expressed in multiple ways by placing the clauses in different orders by means of concatenation. This allows keeping the number of required boilerplates low while at the same time having a high flexibility. (www.requirementsengineering.info, Farfeleder et al. 2011)

The boilerplate repository is available at www.requirementsengineering.info/. Some examples are given in Table 4. For DOORS users, there is a set of scripts that provide support for boilerplates, see section 3.3.1.

3.2.2 CESAR Requirement Specification Languages

CESAR stands for “Cost-efficient methods and processes for safety-relevant embedded systems” and was a European funded project from ARTEMIS joint undertaking (<http://www.cesarproject.eu/>). The project started in March 2009 and was finished in June 2012. CESAR has extended and applied the boilerplate idea above to a number of safety-critical domains, with the intention to formalise the approach using domain ontologies (Dick & Llorens 2012).

The purpose of the CESAR Requirements Specification Languages (RSL) is to capture and formalise requirements throughout the development process. CESAR provides multiple languages instead of a single one (Figure 6). Boilerplate and the pattern-based RSL are textual semi-formal languages, providing templates to engineers that constrain the writing of requirements. (CESAR 2010)

Guided Natural language RSL retains the benefit of free text since it does not introduce additional constraints on requirement statements or require additional training. It can be achieved with a dictionary. Domain-specific and requirements-related terms can be highlighted and the elicitation process enhanced with a content assistant that provides the analyst with related and specific choices of domain terms from a dictionary. (CESAR 2010)

Boilerplates are semi-complete requirements that are parameterized to suit a particular context in a way similar to (Hull et al. 2002). Boilerplates can be linked with domain ontologies that contain commonly agreed domain or company-specific terms. The second boilerplate enforcement mechanism is a set of predefined structures. The pattern-based RSL uses an even stronger formalism with fixed semantics. As a result, patterns constrain users in a stronger way in writing requirements, but allow numerous automatic analysis techniques to be used. (CESAR 2010)

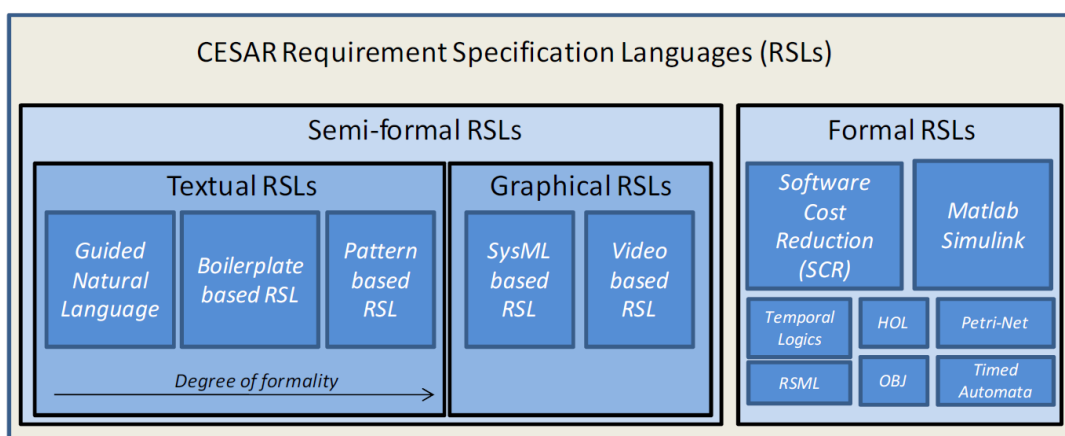


Figure 6. Overview of CESAR Requirements Specification Languages (CESAR 2010).

CESAR uses about 35 boilerplates for expressing user capabilities, functional system requirements and constraints. For example, a combination of BP2 and BP35 leads to

The <system> shall be able to <action> <entity> at least <number> times per <unit>

The original set of boilerplates by Jeremy Dick et al. has been slightly changed. Enhancements allow several entities to be grouped and temporal relationships to be expressed. Interestingly, a temporal state can be achieved, maintained, ceased, avoided, minimized, maximized, increased, decreased and improved. Boilerplates are provided also for optional behaviour (the system may...), goals (in order to...) and negative requirements (example in Figure 7). (CESAR 2010, 2011)

3.1.3.3.10 C-BP23: <system> shall not <action>

Boilerplate expressing a system constraint.

<system>:	(Part of) the system the constraint applies to.
<action>:	A behaviour that shall not happen.
Example:	If its luminescence decreases to below 100 microlamberts , <an operating handle or operating handle cover> shall not <be used>.

Figure 7. A boilerplate expressing a negative functional requirement (CESAR 2011).

The CESAR boilerplates (CESAR 2011) are divided into three types; main, prefix and suffix boilerplates. Main boilerplates can stand alone, whereas prefix and suffix boilerplates must be prepended or appended, respectively. Boilerplates can be classified according to the categories Capability, Capacity (Maximise, Exceed), Rapidity (Minimise, do not exceed), Mode (while, if, for ...), Sustainability, Timelines, Operational Constraints and Exception. This helps ordering the requirements. (CESAR 2011, Johannessen 2012)

In pattern-based RSL, patterns consist of static text elements and attributes being filled in by the requirements engineer. Each pattern has a well-defined semantic in order to ensure a consistent interpretation. Patterns allow the writing of natural sounding requirements while being expressive enough to formalize complex requirements. CESAR defines 25 patterns organized into categories concerning functions, probability, operating mode, safety (Figure 8), timing and architecture. A functional pattern describing the causality between two events does look like this:

[whenever *Event1* from *subject1* is received] *subject2* [does not] emit[s] *Event2* [within *Interval*]

Phrases in square brackets are considered optional, and italic printed elements represent attributes that have to be filled in. (CESAR 2010, 2011)

Pattern 16:	<i>Failure shall be detected with probability p</i>
This pattern is used to specify probability of detecting a failure within the system, e.g. when describing systems that contain BITE (build-in test equipment) or failure mitigation features.	
Natural Language Requirement: A hydraulic system failure shall be detected in 2 of 100 cases	
Pattern based RSL: <i>HydraulicSystemFailure</i> shall be detected with probability 0,02	

Figure 8. Description of a safety pattern (CESAR 2010).

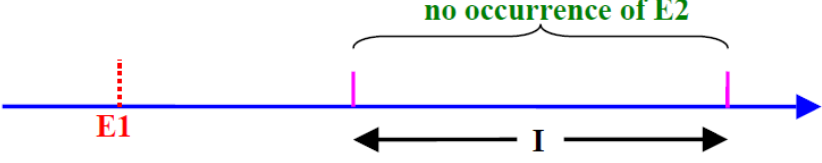
Pattern Id	P3
Name	whenever [E1] occurs [E2] does not occur during following [I]
Description	<p>Following every occurrence of E1, E2 will not occur during the following occurrence of the interval I. The occurrence-instances are triggered by the occurrence of E1 and ends with the termination of the following occurrence of I. Note that it is not possible to use a term like $\neg E2$ since the negation of events is not defined in our formalism (this is in contrast to conditions).</p> 
Usage	Iterative
- Example: 40 sec. minimal delay between trains: - whenever [Tin] occurs [Tin] does not occur during following (40sec)	
<i>Derived Patterns</i>	
[E1] cannot occur before [E2]	whenever [StartUp] occurs [E1] does not occur during following [StartUp,E2]

Figure 9. Example of a contract assertion pattern (SPEEDS 2008)

Moreover, CESAR has considered the contract-based approach in requirements engineering, in particular the ideas developed in the SPEEDS project for embedded system design (<http://www.speeds.eu.com/>). Different from the legal binding, the term “contract” refers in software engineer’s literature to the set of specifications that describe what a system should guarantee under some assumptions (CESAR 2010). In other words, a contract-based requirement has two parts: 1) an assumption about the operating environment not controllable by the system and 2) a promise saying what the system will realize given that the assumption is satisfied (CESAR 2011). Also SPEEDS (2008) defines a set of most frequent patterns that are used in contracts specification. Figure 9 gives an example. They might be used as requirement templates also.

In addition to the issues described above, the CESAR (2011) project has studied several topics relevant to the SAREMAN project. In particular, it has considered generic failure modes, preliminary hazard analysis on the basis of system requirements, model-based requirements modelling, semi-automatic formalisation of requirements and use of behavioural patterns (Figure 10) for model checking (see section 3.5). Moreover, CESAR has developed tools for requirements modelling and analysis. One of them, DODT, is introduced below in section 3.3.4.

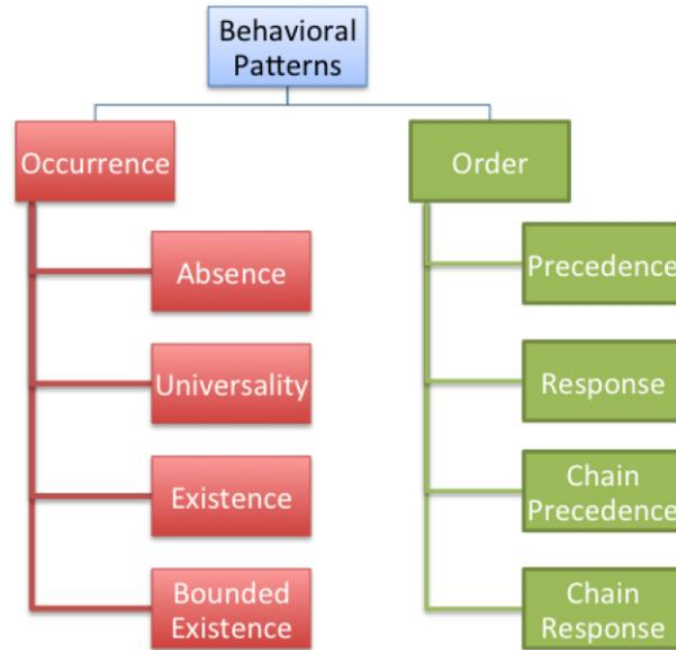


Figure 10. Classification of behavioural patterns (CESAR 2011).

3.2.3 Easy Approach to Requirements Syntax (EARS)

EARS was developed by Mavin et al. (2009, 2010) at Rolls-Royce Control Systems. In EARS, the requirements are based on five sentence templates: ubiquitous requirements, event-driven requirements, state-driven requirements, unwanted behaviour requirements and optional requirements (Figure 11). Template types are identified by keywords 'when', 'while', 'if-then' and 'where'. EARS does not impose a strict structure to the elements that are filled in. Basic templates can be combined to create compound statements.

EARS has been successfully applied to a variety of complex, safety-critical systems, e.g. aero-engine control systems. The method has been developed primarily for stakeholder requirements, as opposed to technical system requirements. The hypothesis of EARS is that a small set of simple requirement structures is an efficient and practical way to enhance the writing of high-level stakeholder requirements.

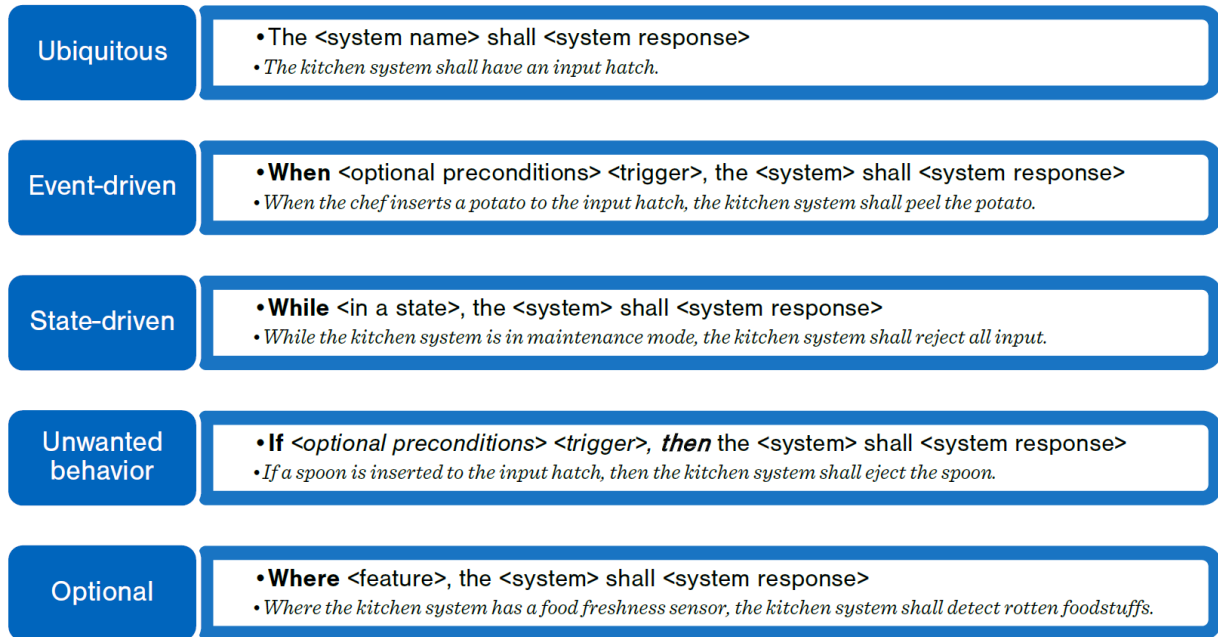


Figure 11. Sentence types of EARS (Easy Approach to Requirements Syntax)(SAREMAN 2013b).

3.3 Tools for requirements definition and analysis

This section describes some examples of commercial and experimental software tools.

3.3.1 DOORS

Rational DOORS, a product of IBM, is one of the most well recognised requirements management tools available (Beatty et al. 2011). Like other common requirement authoring tools, DOORS (Figure 12) basically emulates a word processor but offers additional features (OMG 2013). In a 2011 evaluation of requirement management tools (Beatty et al. 2011), the consulting company Seilevel noted that DOORS is “an excellent all-around tool”, and ideal for large system projects that need to handle a large volume of data. Traceability, queries, reporting, and access control features provide flexibility for many types of work. The limitations include a somewhat out-of-date user interface, with some functionality buried deep within menus. Also, emphasis is on requirements traceability and versioning rather than requirement *modelling*, making DOORS a tool more suitable for software projects than systems engineering.

For exchanging and processing requirements, DOORS provides several mechanisms, for example:

- The Rational DOORS eXtension Language (DXL) is a C or C++ like scripting language that allows users to create plugins that control and extend the functions of DOORS.
- Rhapsody Gateway add-on can be used to export requirements from DOORS (as well as various other sources) to Rhapsody as SysML artifacts, or upload Rhapsody model elements to DOORS.

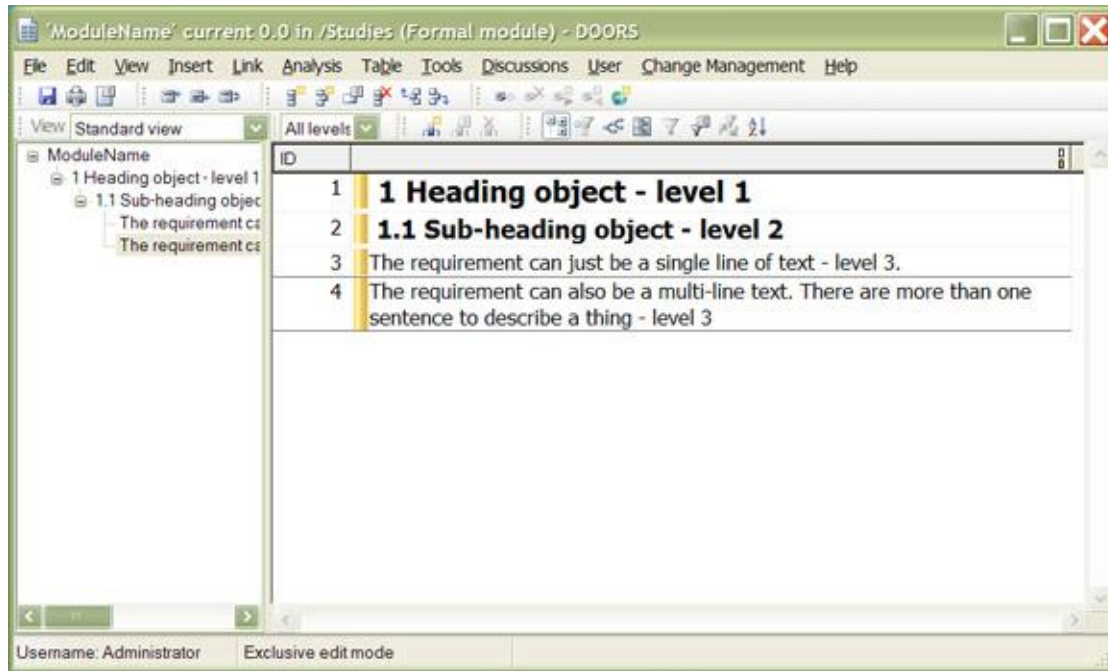


Figure 12. DOORS documents have a tree-like data structure (Image source: www.ibm.com).

As a data format for exchanging requirements, DOORS supports the Requirements Interchange Format (RIF), an OMG standard intended to facilitate the exchange of requirement information between various tools and companies. RIF is an XML extension supporting, e.g., hierarchical document structure, unique identifiers, XHTML based text formatting, and access restrictions (OMG 2013). The requirement attributes can also refer to external objects with binary content (e.g., image files).

Basically DOORS requirements, as well as the requirement clauses in RIF documents, are free text. However, there are DXL scripts available at www.requirementsengineering.info that support for the boilerplates by Hull, Jackson and Dick (2002) introduced in section 3.2.1. These scripts, dating back to 2002, should allow requirements to be constructed from boilerplates (Figure 13). This tool shows the complete requirement text with attributes highlighted in blue. The user can edit the attributes by double-clicking them and then either entering a new value or selecting a value from those used in other requirements.

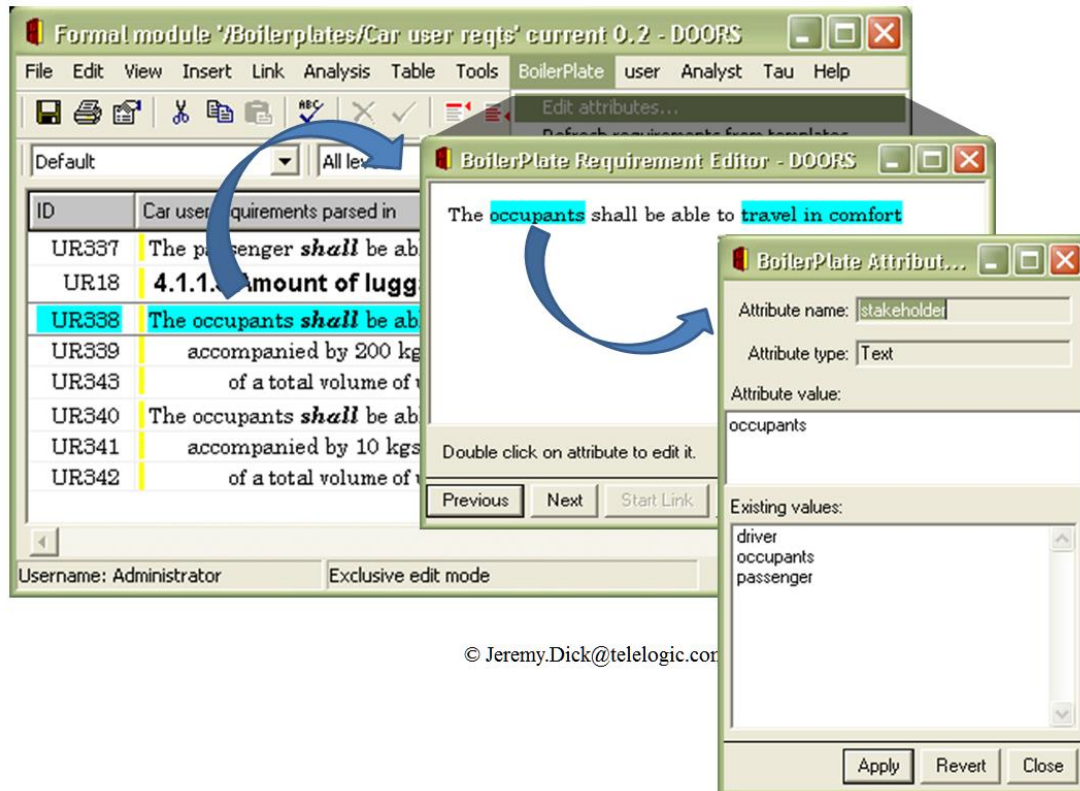


Figure 13. Requirement Boilerplates in DOORS (adapted from slides by Jeremy Dick, www.requirementsengineering.info)

3.3.2 Requirements Authoring Tool (RAT)

The Reuse Company (<http://www.reusecompany.com/>) is a European IT company with the mission to promote information reuse by offering processes, methods, tools and services. Its main product, the Requirements Quality Suite (RQS) consist of several tools, in particular:

- Requirement Authoring Tool (RAT²) to assist authors in creating or editing requirements.
- Requirements Quality Analyzer (RQA) to setup, check and manage the quality of a requirements specification.
- knowledgeMANAGER to manage the knowledge around a requirements specification, e.g. the ontology it is based on, the structure of requirements (boilerplates) and the communication between authors and domain architects.

The Requirements Quality Analyzer (RQA) for DOORS enables managing the quality of requirements specifications in projects using DOORS. It assesses the quality of a DOORS repository by Natural Language Processing (NLP) techniques to extract quality metrics for readability, ambiguity, incompleteness, domain terms and boilerplates matching with the written text. The ontology defined with the knowledgeMANAGER stores the information needed for requirements authoring and quality analysis (Figure 14). (Reuse 2013)

² Note that there has been also another tool development called RAT - Requirements Analysis Tool (<http://rat.fbk.eu/>) originally developed in the PROSYD project funded by the European Commission. Since then, an upgraded version named RATSy (Requirements Analysis Tool with Synthesis) has been released (<http://rat.fbk.eu/ratsy/>).

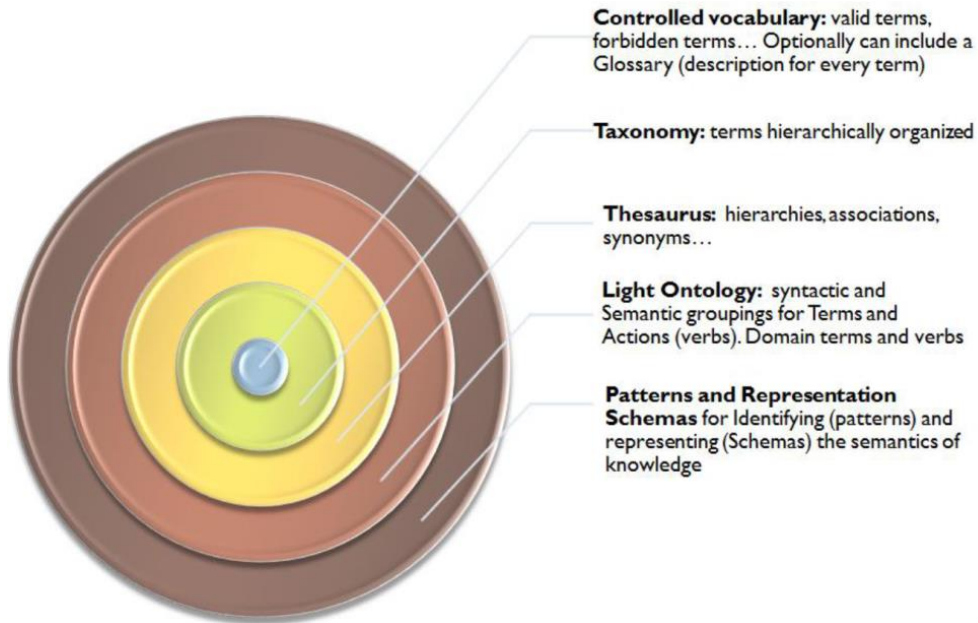


Figure 14. RQA ontology (Reuse 2013).

The Requirement Authoring Tool (RAT) uses a set of boilerplates but allows the user to type free requirement text (Figure 15). The quality frame on the right side shows the quality of the written requirement. Based on the dictionary and the quality assessment results the text box shows spelling errors and terms that should be considered for re-authoring.

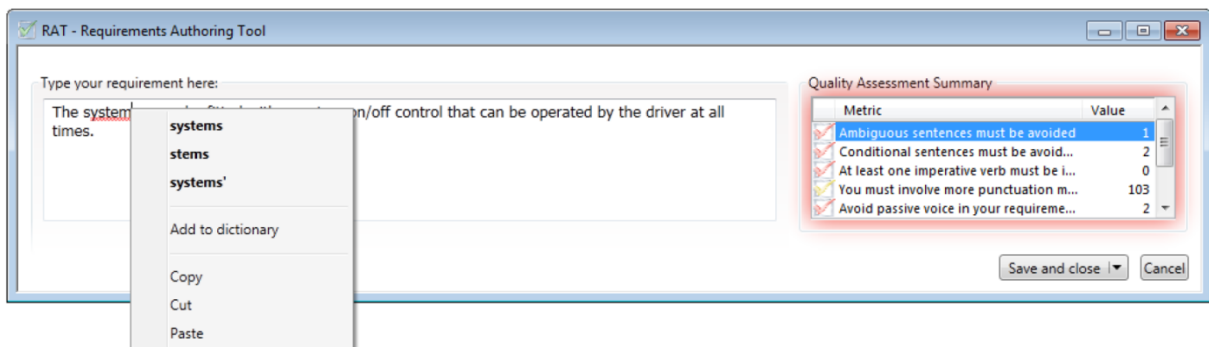


Figure 15. Typing free text in RAT (Reuse 2013).

While allowing the author to write free text, it is also possible to first select a specific boilerplate (Figure 16). RAT shows its structure and an example of use. The author then selects the right terms from the suggestions. RAT shows the feasible boilerplates matching with the written text. Furthermore, users can suggest new boilerplates, or changes to the existing ones.

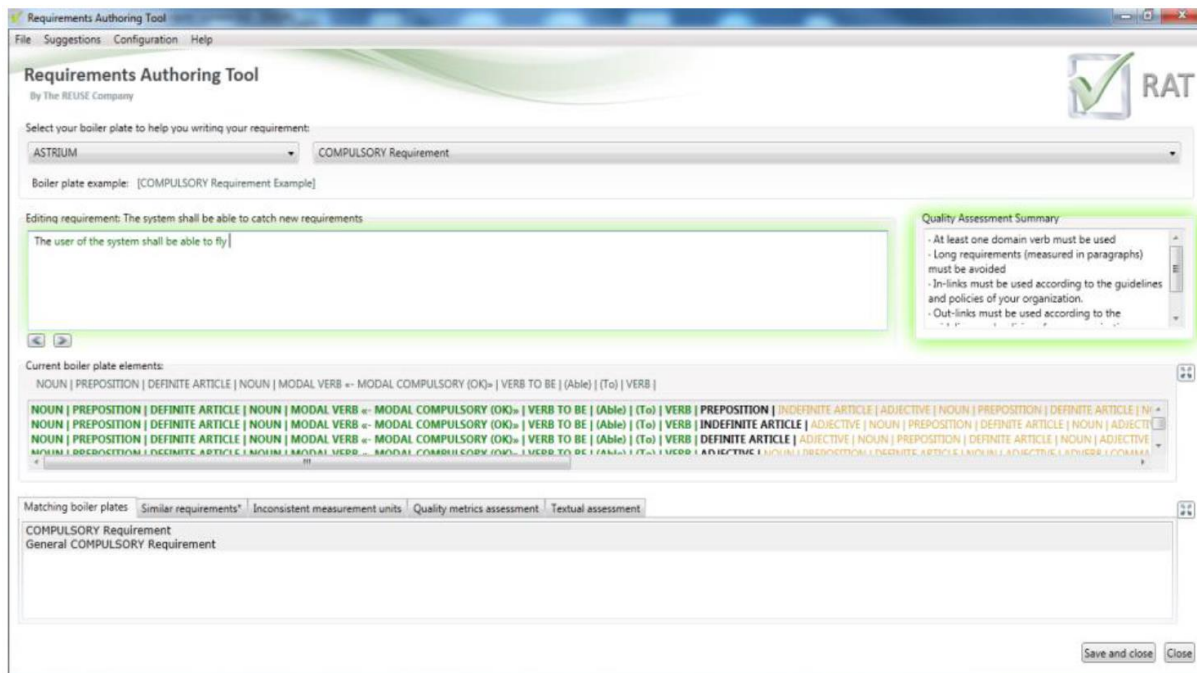


Figure 16. Editing requirements with boilerplates (RAT flyer).

RAT uses the same metrics as RQA but focuses on individual requirements and has been designed as an authoring assistant. It provides quality feedback on the fly and detects inconsistencies, coupling requirements, ambiguous requirements, non-atomic requirements, use of the wrong verb tense, mode or voice, and inconsistent use of measurement units. RAT is connected to the Requirements Management tools, e.g. IBM Rational DOORS, Microsoft Excel. (Reuse 2013)

RAT allows the user to start typing a requirement, and simultaneously determines which of a palette of templates are applicable, guiding the user on permissible terminology. The combination of these properties allows the requirements author to write requirements in an assisted way, as a counterpart to the “fill the gap” approach to placeholders. The possibility of not selecting a boilerplate before typing a requirement leads to the tool being considered as an assistant, and users may feel their creativity being less constrained. At present, the tool comes equipped with about 70 templates that can be combined in various ways. (from Dick & Llorens 2012)

3.3.3 Context RDS

Context RDS from the developers at Rolls-Royce is an experimental toolkit that develops the concept of statement templates towards the use of domain-specific ontologies. The tool constrains how authors are able to instantiate the templates. This ensures, for instance, the consistent use of terms and units, and prevents the creation of nonsense requirements (Dick & Llorens 2012). The RDS template structure shown in Figure 17 is close to what we are thinking about in SAREMAN. However, it is unclear whether Rolls-Royce intends to develop this prototype further and how it would be related to the EARS guidelines described above in section 3.2.3.

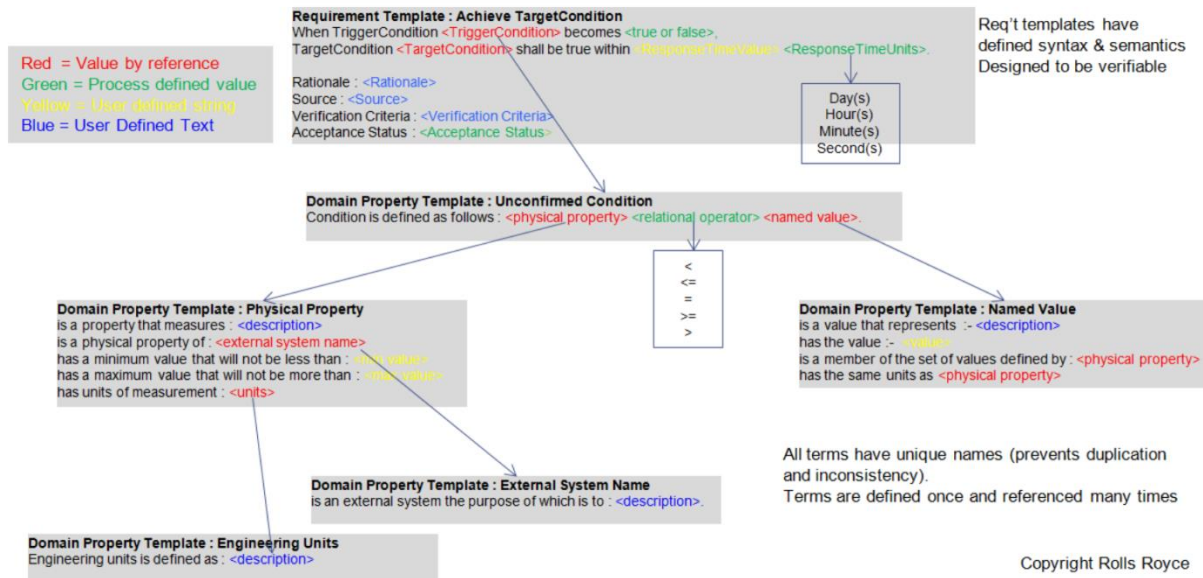


Figure 17. Context RDS template structure (Dick & Llorens 2012).

3.3.4 CESAR DODT

DODT is a tool developed in the ARTEMIS project CESAR (<http://www.cesarproject.eu/>, see section 3.2.2 above) to write and manage boilerplate requirements. The tool allows the creation and customization of ontologies, boilerplate database and a requirements specification. The requirements editor is shown in Figure 18. The tool monitors whether the attributes are in accordance with the ontology, indicates errors and allows the requirements engineer to create a concept. It is also possible to create or import ontologies. Moreover, DODT enables the user to create and edit boilerplates. (Farfedler et al. 2011, Johannessen 2012)

One purpose of the domain concepts and generic failure modes in DODT is to support hazards analysis at an early stage of design. For example, let's consider the requirement:

R4: The control system shall control water level using feeding pump.

By identifying the participating elements (control system and feeding pump), DODT is able to automatically generate a table of elements and their failure modes. The failure effects, their causes and corrective actions need to be filled-in by a safety expert. KROSA is prototype tool developed in CESAR for the safety analysis (FMEA). It is based on natural language processing (NLP, see section 3.1), hazard analysis ontologies and Case Based Reasoning (CBR) on a library of previous (similar) hazard analysis results. (CESAR 2011)

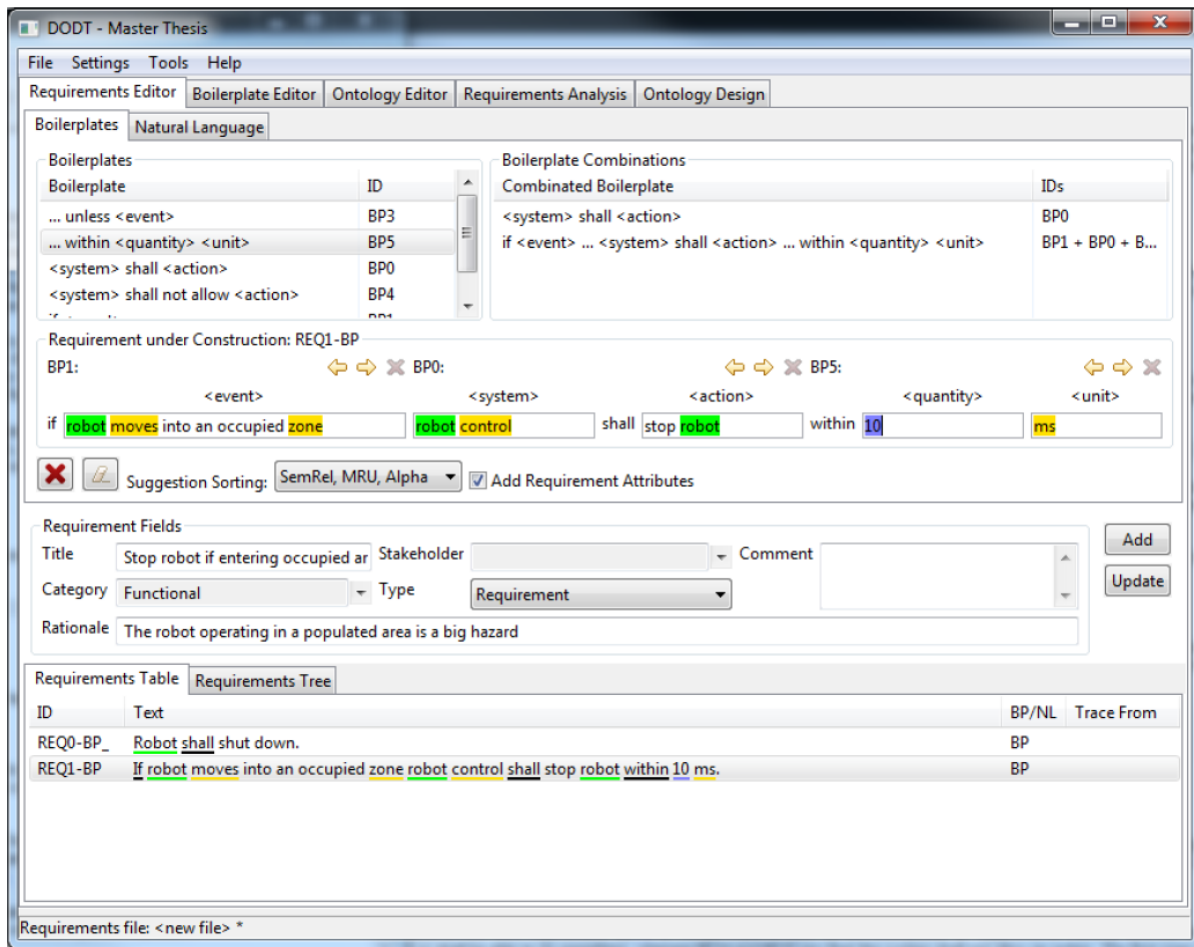


Figure 18. DODT Requirements editor (Johannessen 2012).

An example of using CESAR tools in developing automotive systems is described by Armengaud et al. (2012). In the requirements engineering phase requirements are formalized from natural language text to semi-formal boilerplates using DODT and finally to formal patterns using the PatternEditor (Figure 19). The requirements are stored in the RequisitePro tool. The CESAR Meta-Model (CMM) API would allow integration to other RM tools like DOORS without changes to the application. Mistakes can be detected early by running the ontology based analyses or formal requirement checks. In system design and safety analysis, the EAST-ADL2 architecture description language is used for modelling automotive embedded systems with the Papyrus tool. Requirements are linked to system elements. Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) support safety design by automating FTA and FMEA. Test cases are derived from requirements and executed automatically. Traceability is maintained between requirements, system elements, test cases and test results. This example illustrates nicely the fact that many kinds of tools need to be integrated today in order to cover the whole design process. For instance, requirements management tools don't support modelling of system architecture. On the other hand, current design tools have no support for requirements. Therefore, tailored solutions are needed to exchange information between them.

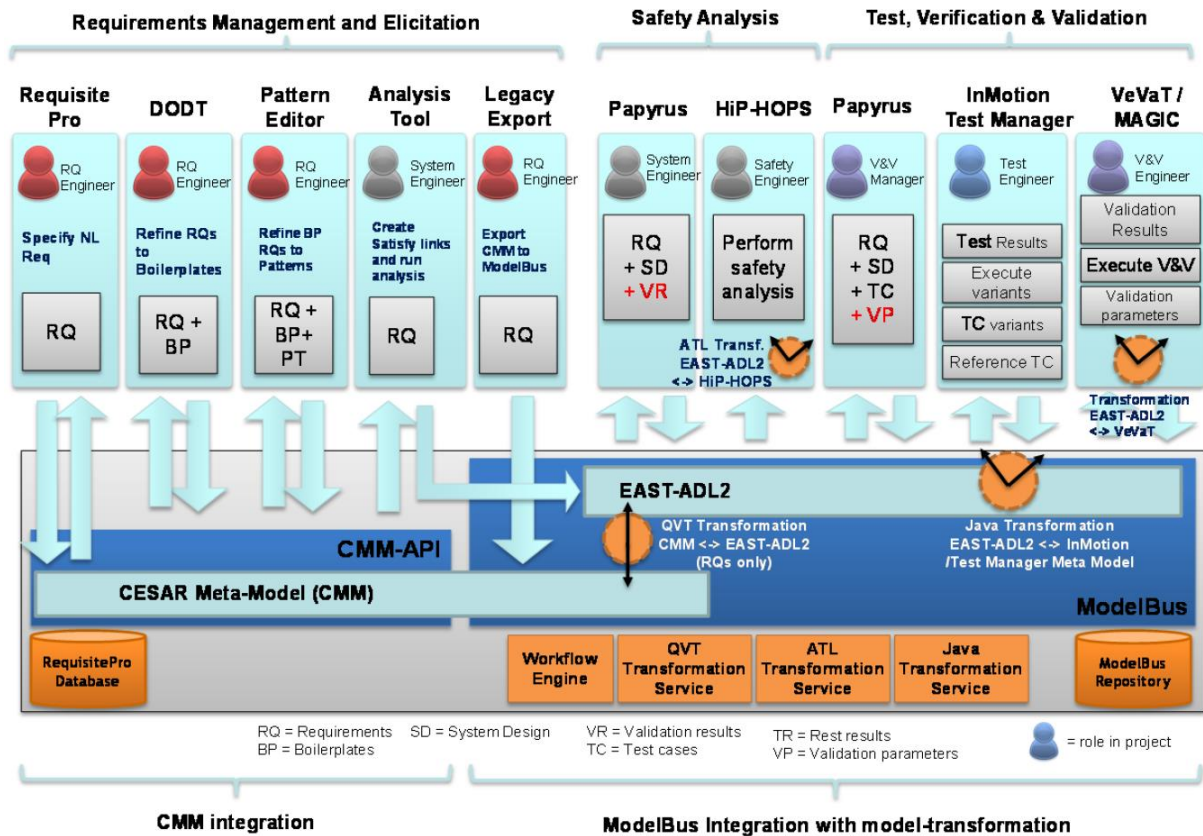


Figure 19. Proposed tool-chain for embedded automotive systems (Armengaud et al. 2012).

3.3.5 Attempto Controlled English (ACE)

Attempto is a research project of the University of Zurich with the objective to develop Attempto Controlled English (ACE) and its tools (see <http://attempto.ifi.uzh.ch/site/>). The intention is to help professionals who want to use formal methods but may not be familiar with them. While originally designed for software specifications, ACE has in the recent years been extended with languages and applications of the Semantic Web (Kaljurand & Kuhn 2013).

ACE allows users to express texts precisely with the terms of their application domain. The ACE Editor helps users to construct correct ACE texts, i.e. one or more sentences that can refer to each other. ACE is a *Controlled Natural Language* (CNL, see section 3.1) with a precisely defined subset of English that can automatically and unambiguously be translated into various forms of first-order logic. ACE uses a built-in lexicon with approximately 100 000 entries, but users can import additional, domain-specific lexicons. ACE's grammar and meaning are defined in a set of construction and interpretation rules. To avoid ambiguity, certain constructs are not part of the language, and all others are interpreted deterministically. Attempto uses Kamp's rules of discourse representation (<http://plato.stanford.edu/entries/discourse-representation-theory>) to resolve the referents of pronouns and definite noun phrases. This is different from, e.g., CLCE that uses temporary names to avoid ambiguity (<http://attempto.ifi.uzh.ch/site>, Fuchs et al. 2008, Kaljurand & Kuhn 2013)

ACE is supported by a number of tools, for example a parser and reasoner. The former takes an ACE text and optionally a user lexicon as input, and generates a large number of various outputs, for example parse trees and formal "discourse representation structures" (DRS). The Attempto Reasoner RACE offers three operating modes: consistency checking, proving and query answering. Applications of ACE include software and hardware specifications,

data bases, agent control, medical regulations and ontologies. Furthermore, ACE can serve as a natural language interface to semantic web applications. (Fuchs et al. 2008)

As a recent extension, Kaljurand and Kuhn (2013) describe a version of the AceWiki, a CNL-based semantic wiki engine that uses ACE as the content language and OWL as its underlying semantic framework. The natural language grammar is implemented on top of the Grammatical Framework (GF), a functional programming language for building multilingual applications (see <http://www.grammaticalframework.org/>). GF facilitates semantic reasoning and bidirectional automatic translation between ACE and a number of 15 European natural languages. Additionally, the approach allows for automatic translation, e.g., into the Web Ontology Language (OWL). The grammar describes mapping between abstract logical expressions and the corresponding sentences in various languages. This mapping is bidirectional — strings can be parsed to abstract trees, and trees linearized to strings (Figure 20).

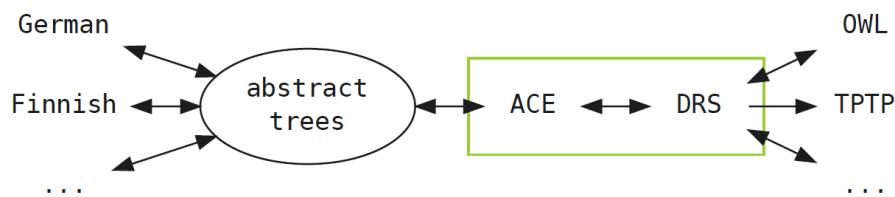


Figure 20. Bidirectional mapping between a formal language like OWL or TPTP (related to automated theorem proving) and a natural language like Finnish (Kaljurand & Kuhn 2013).

Attempto seems to be a good example of advanced natural language processing environments indicating that NLP tools might be practical in engineering tasks. Some of its features may even be too sophisticated for our purposes, i.e. for writing well-formed requirements and other statements needed in control system design. In particular, bidirectional transformations and support for several languages could be desired features of our solutions too. For example in model checking, there might be several alternative output formats corresponding to a natural language statement. In the other direction, there is an obvious need to explain/verbalise complex formal presentations like temporal logic.

3.3.6 ReqUirements BoileRplate sanlty Checker (RUBRIC)

When boilerplates are used as guidance to the requirement author, there is a need to verify that the requirements conform to the boilerplates. Doing this manually is laborious. A prototype tool RUBRIK by Arora et al. (2013a, b) uses natural language processing and common vocabularies in a way that doesn't require exact matches to a boilerplate and terms in a glossary.

Support for boilerplates already exists in commercial RE tools. Many of them assume that all terms (a domain ontology) have been defined in a glossary. However, building a glossary often concludes only after requirements have been written and may even remain incomplete throughout the whole project. The typical approach does not work when the glossary terms are unknown.

RUBRIC (ReqUirements BoileRplate sanlty Checker) provides automation for checking conformance to boilerplates using a *Natural Language Processing* (NLP) technique called Text Chunking. Noun and verb phrases (see section 2.6) provide a suitable level of abstraction for checking conformance. RUBRIC further provides diagnostics to highlight potentially problematic syntactic constructs in the requirement statements (Figure 21, Figure 23). A text chunker is a pipeline of NLP software modules decomposing a sentence into non-overlapping segments, e.g. to noun and verb phrases (Figure 22). Its effectiveness is not compromised even when the requirements glossary terms are unknown. RUBRIC was

publicly released in June 2013 and is available at: <http://sites.google.com/site/rubricnlp/>. It uses an open-source natural language processing framework called GATE, see gate.ac.uk/.

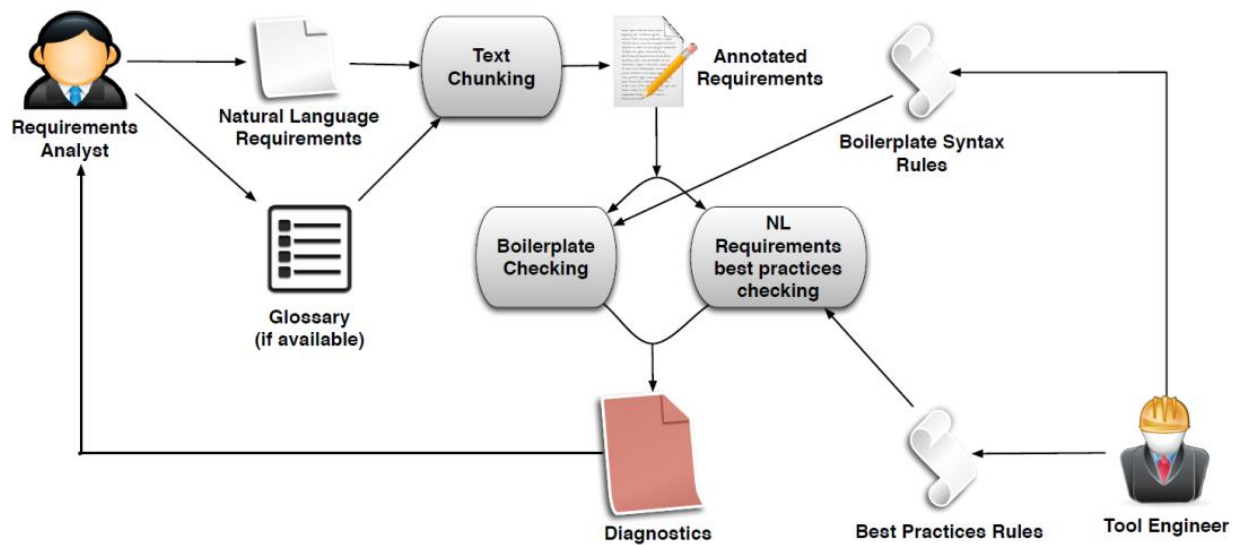


Figure 21. Overview of RUBRIK (Arora et al. 2013b).

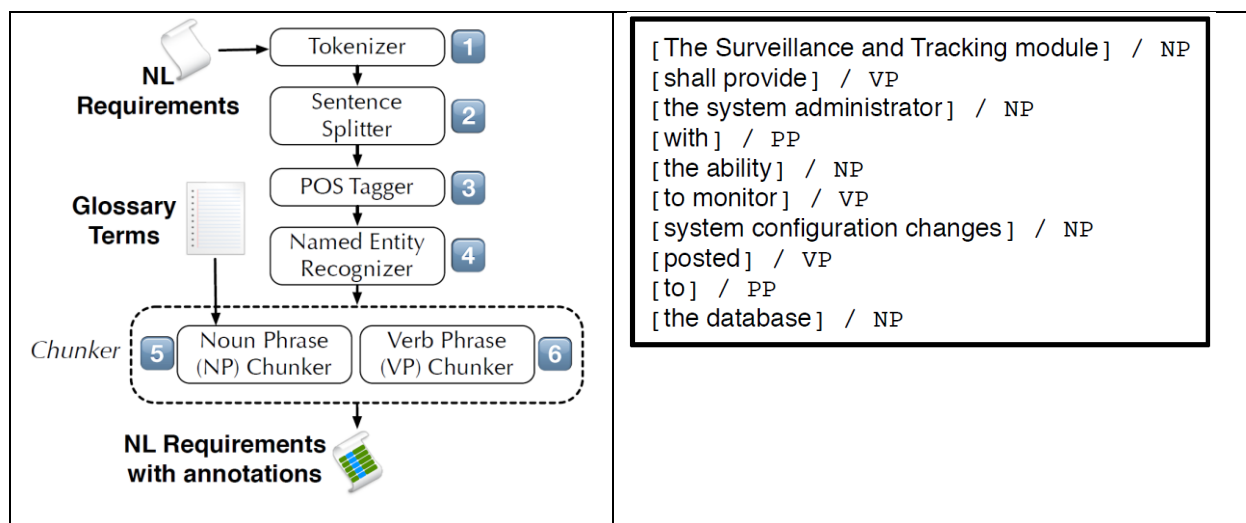


Figure 22. Text chunking steps and the resulting list of phrases (Arora et al. 2013a).

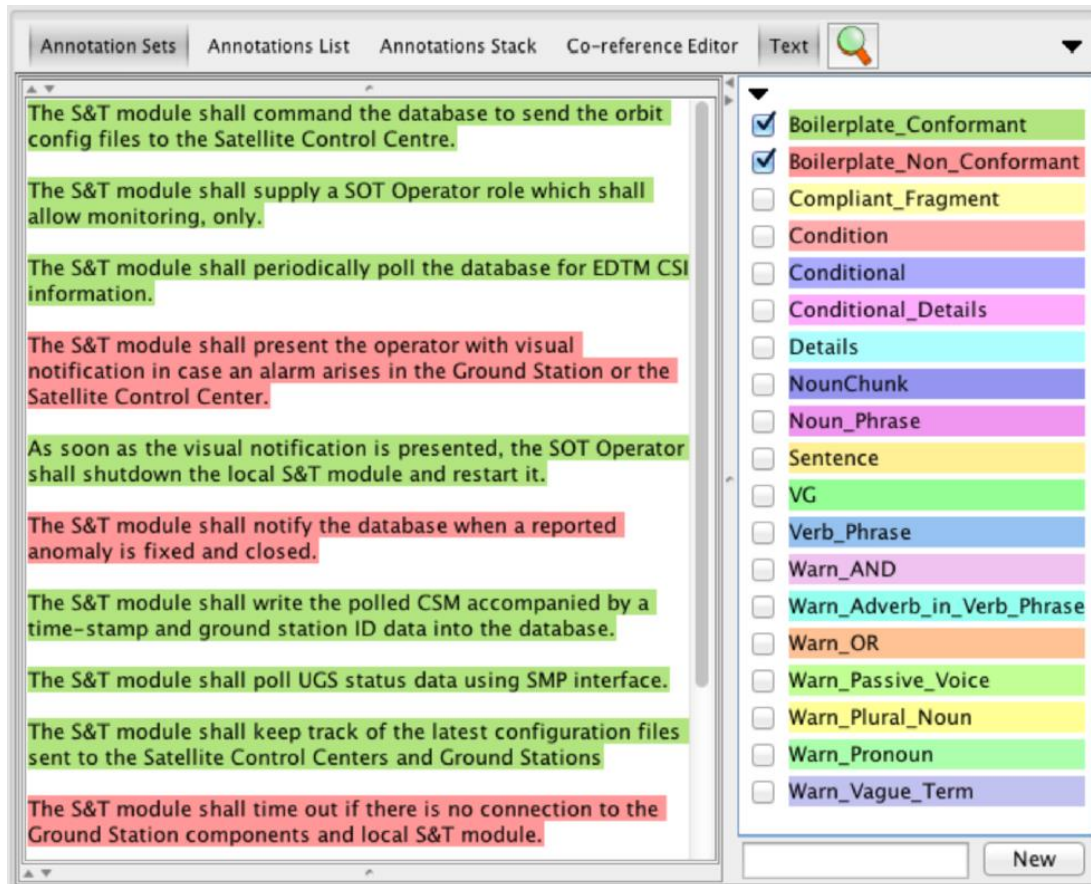


Figure 23. Markup generated by RUBRIC (Arora et al. 2013a).

As claimed by Arora et al., a key advantage of RUBRIC is that it yields good results even in early stages of requirements writing, where a glossary may be unavailable. Their approach doesn't apply to model checking but might be valuable for writing stakeholder and system requirements. In addition to missing terms, the types of requirement statements at this level are diverse and therefore can't be forced into a fixed format. Detecting common language terms (e.g. Princeton WordNet) and phrases and guiding the author to combine them in recommended ways might be a practical solution for our purposes.

3.4 Testing automation

Testing is a critical and expensive task in industrial automation. Parts of the control application are first tested separately in *module tests*. In addition to individual control loops, safety critical and widely re-used library functions (function blocks) should be verified as early as possible. The *Factory Acceptance Test (FAT)* then demonstrates that the integrated control system is in accordance with the specifications (IEC 62381 2011). Items to be checked include, for example, documentation and system configuration, system IO and communication, basic control and protection functions, visualization, operation and complex functionality (for example sequence control) as far as possible without the actual process system being present. FAT is performed by the vendor before delivery to the site of installation. The customer should witness the test activities and in some cases carry out some parts of the FAT itself.

Requirements are closely related to system and software testing during various life-cycle phases. The purpose of *test cases* is to *verify* that the implementation satisfies the requirements, regulations or specifications. Therefore, test cases should be derived from and traced back to requirements and design specifications. In some cases, the transformation is relatively straightforward but sometimes several complex test scenarios are needed to verify one requirement. Moreover, one test case can be designed to verify several requirements.

One problem is that tests are defined in terms of elements in the implementation, e.g. variables and IO channels, while requirements describe the system on a more abstract level.

Testing is a major issue in several software development approaches. *Test-driven development* (TDD) is a variant of agile software development that relies on a short development cycle. First the developer writes an automated test case that defines a desired improvement or a new function and then produces the minimum amount of code to pass that test (adapted from http://en.wikipedia.org/wiki/Test-driven_development).

Behaviour-driven development (BDD) was developed on the basis of TDD by Dan North (<http://dannorth.net/introducing-bdd/>) as a response to certain issues encountered teaching test-driven development. BDD uses natural language as a ubiquitous communication mean to describe the acceptance tests by means of scenarios. In fact, the natural language ensures a common understanding of the system to be developed between all members of the project – particularly between the designers and the stakeholders. Currently, the BDD approach is still under development. However, there are various toolkits supporting BDD, such as JBehave, Cucumber and RSpec.

In BDD plain text descriptions of features, user stories and scenarios make use of pre-defined templates. Typically *user stories* are specified using the following template:

```
[StoryTitle] (One line describing the story)
As a [Role]
I want a [Feature]
So that I can get [Benefit]
```

A user story is refined as a set of *scenarios* each describing how the system that implements a feature should behave when it is in a specific state and an event happens (Solis & Wang 2011). The template for writing scenarios consists basically of the triplet Given-When-Then:

```
Scenario 1: [Scenario Title]
Given
    [Context]
    And [Some more contexts]....
When
    [Event]
Then
    [Outcome]
    And [Some more outcomes]....
```

Both defining the tests and executing them manually is expensive in terms of money and time. So, they should be automated. In *Model-Based Testing* (MBT) test cases are typically generated from system specifications represented in a (semi-)formal model, such as UML. Many approaches, some of them shortly reviewed by Carvalho et al. (2013), start from requirements written in a *Controlled Natural Language* (CNL) by transforming them into an internal, more formal form (e.g. Object Constraint Language, temporal logic or business rules) and then further to test case definitions. The idea is to provide a specification that is close to natural language but also formal enough to be processed by computers. The final result of this process can be, for example, directly executable test code or a higher-level test script that can be processed by a test automation environment.

For instance, *keyword-driven testing* can be used to automate the testing effort. It is based on pre-programmed actions (keywords) that read and write the variables of the System Under Test (SUT). A command sent to the SUT consists of a keyword and optional parameters. For example, Hametner et al. (2012) have used keyword-driven testing to PLC-based industrial automation. Their keywords include, for example:

- startConn: establishes a connection to the SUT
- set: sets a new value to the variable in the SUT
- sleep: pauses the test for a specified time
- get value: reads the value from a SUT variable
- check: compare the value with the expected

Robot Framework (<http://robotframework.org/>) is one of the available tools for automated testing. As an example, let's look at the following statement requiring that the level in a tank rises soon after a pump is started:

```

    Given that ( T300_Empty == TRUE ),
    when ( P201_CtrlOut is set to TRUE ),
    condition ( T300_Level > 10 ) becomes TRUE within 5 minutes.
  
```

When transformed into a keyword-driven test script in Robot Framework style, the test case might look like this:

```

| Read All Variables |
| Should Be True | B300_Empty == True) | |
| Should Not Be True | (T300_Level} > 10) |
| Write Variable | P201_CtrlOut | True |
| Sleep | (5 * 60) - 1 |
| Read All Variables |
| Should Be True | T300_Level} > 10 |
  
```

In software engineering, we can find several attempts to derive executable test cases from semiformal requirements written in a CNL (e.g. Bacherler et al. 2012, Carvalho et al. 2013). As an example from the control engineering domain, the Template Based Natural Language Specification (TBNLS) (Esser & Struss 2007) was a (preliminary) CNL approach for functional tests of control software for passenger vehicles. The aim was to build a tool that supports an existing work process without major revisions. Their proposed solution offers a natural-language-template-based interface for acquiring software requirements. The language is defined by 15 templates that provide a mapping to propositional logic with temporal relations. The approach was based on trying to confirm that faulty behaviours are not possible. Potential faulty behaviours are generated from formalized versions of the requirements by a number of (transformations. The fault types are defined mainly to match the intuition of “what may go wrong” behind manually generated test cases. Two of the most obvious fault types are: 1) The start conditions are satisfied, but the consequence does not occur; and 2) the condition is not satisfied, but the consequence occurs, anyway. Also Schnelte (2009) represents a related experiment based on natural language templates. In addition, he used a planning algorithm to create test sequences for positive and negative failures.

3.5 Model checking and property specification patterns

Model checking (Clarke et al. 1999) 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.

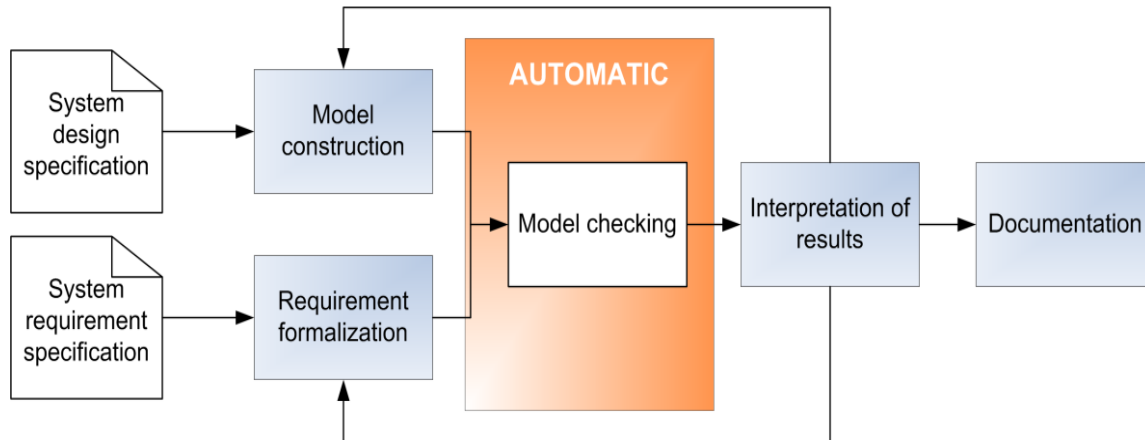


Figure 24. The overall model checking process involves several steps where manual work and specific expertise are required. In this paper, we focus on requirement formalisation.

The overall process of performing model checking is illustrated in Figure 24. While the actual analysis is performed automatically by the model checker, there are several tasks that require manual work and specific expertise. In particular, the functional requirements need to be formalised using temporal logic for expressing the correct model behaviour. 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 (Tommila, Pakonen & Valkonen 2013). Property specification patterns have been proposed as a link between natural language requirements and formal representations, to support the use of finite-state verification tools such as model checkers. An influential collection of such patterns was first proposed in (Dwyer et al. 1998) and (Dwyer et al. 1999). According to the authors, a specification pattern is a "generalised description of a commonly occurring requirement on the permissible state/event sequences in a finite-state model of a system". The patterns describe some aspect of a system's desired behaviour and provide suitable expressions for the behaviour in a set of formalisms.

Following the example of the "Gang of Four" design patterns (Gamma et al. 1994), each pattern consists of a name, a statement of the pattern's intent, mappings into different specification formalisms, examples of pattern use, and relationships to other patterns (See Figure 25). The patterns are divided into those that require states or events to occur or not to occur, and to those that constrain the order of states or events.

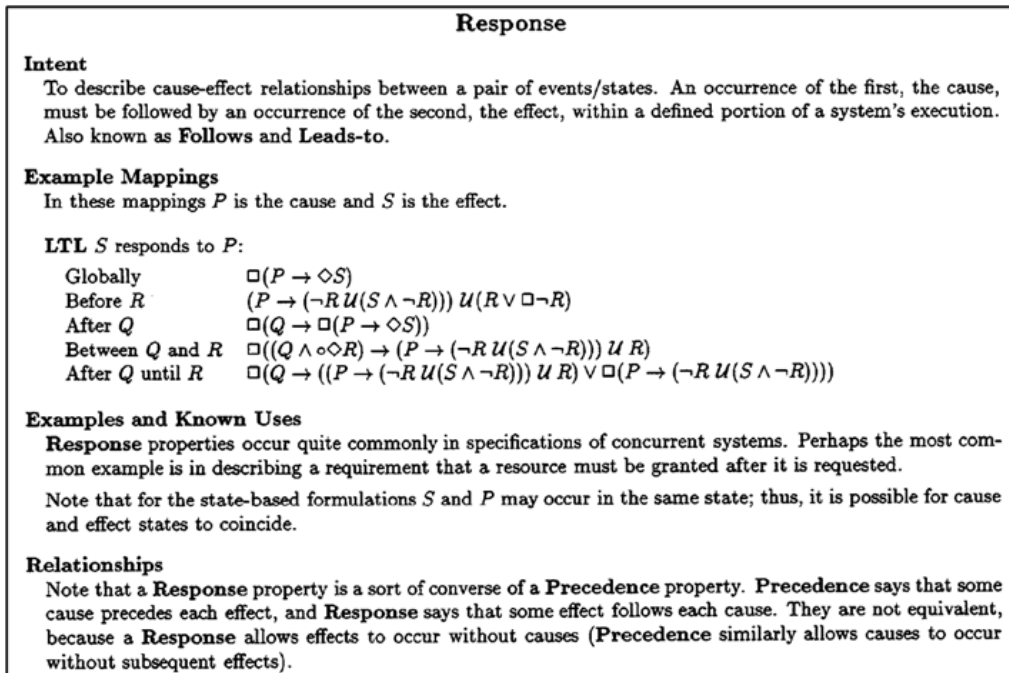


Figure 25. The “Response” specification pattern provides formal expressions for the statement “ P leads to S ”. For simplicity, only the Linear Temporal Logic formulas are listed here. (Modified from (Dwyer et al. 1998)).

As seen in Figure 25, each pattern has a scope that specifies the extent of model execution over which the pattern must hold, in terms of system states – for example “Globally”, in all system states. There are five basic scopes: global, before, after, between, and after-until. The portions of execution designated by each scope are illustrated in Figure 26, with Q and R being placeholders for propositions that describe some state or event.

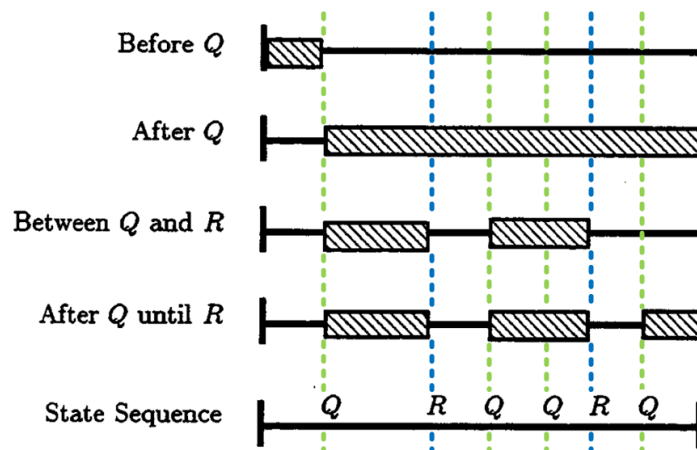


Figure 26. The scope specifies the extent of the model execution over which the pattern must hold. The dotted vertical lines mark the state where the associated proposition Q or R becomes true. (Modified from (Dwyer et al. 1998))

In (Dwyer et al. 1999), the authors claim that according to their experience, 92% of collected requirements matched one of their patterns, the most popular ones being *Response* (“ P leads to S ”), *Universality* (“ P is true”) and *Absence* (“ P is false”).

3.6 Property Specification Language (PSL)

Property Specification Language (PSL) (IEC/IEEE 2012) is a standard language for the formal specification of properties or assertions about hardware designs. Although compatible with hardware design languages such as VHDL, Verilog, and SystemC, it can just as well be used to formalise requirements for I&C software.

Formally, PSL is an extension of standard temporal logic languages Linear Temporal Logic (LTL), and Computation Tree Logic (CTL). PSL is designed to be easier to learn, write and read, with the aim that a PSL specification is “human readable” (Eisner & Fisman 2006). In addition to syntactic sugaring, PSL offers very simple and convenient expressions for, e.g., assertions on sequences of system behaviour. Expressed with LTL, such sequences easily lead to convoluted and incomprehensible formulas.

PSL consists of four layers:

- The *Boolean layer*, consisting of propositions whose value is either true or false.
- The *temporal layer* consists of properties that describe relationships between Boolean expressions over time.
- The *verification layer* is used to tell verification tools what to do with the properties described by the temporal layer. `assert` is a directive to verify that a property holds. As a property merely describes behaviour, it does not specify whether it is a “good” (or a “bad”) thing that the property holds, but an assertion sets a requirement. `assume` and `restrict` instruct the verification tool to constrain the verification so that the given property holds. `cover` directs the verification tool to check if a certain execution path was covered by the verification.
- The *modelling layer* is used to model behaviour of inputs (that is, to specify a sort of “environment model” that is not part of the system model being verified, but limits the values that the system model inputs can have to “realistic” combinations and sequences), and to model auxiliary signals that are not a part of the system but needed for verification.

The following PSL syntax examples are from (Eisner & Fisman 2006): The property “signal *a* leads to signal *b* in the next processing cycle” is stated as:

```
assert always (a -> next b);
```

This is an LTL style expression, with syntactic sugar (`always` for *G*, `next` for *X*). Also included is the `assert` keyword. Furthermore, a PSL variant of the next operator allows us to conveniently state the property “signal *a* leads to signal *b* from 3rd to 5th following cycle” as:

```
assert always (a -> next a[3:5] (b));
```

The same expression could be stated through nested use of the *X* operator of LTL as well, but the result would be nowhere near as readable.

In addition to the LTL style, properties can be built using SEREs, or Sequential Extended Regular Expressions. SEREs describe single- or multi-cycle behaviour as a series of Boolean expressions. A simple example: The SERE

```
{(req out && !ack) ; (busy && !ack)* ; ack}
```

describes a sequence where `(req out && !ack)` holds in the first cycle, `(busy && !ack)` is then asserted for zero or more cycles, and finally `ack` is asserted. Several different

operators (like $[*]$ for “zero or more”) are provided for controlling repetition. SEREs can consist of other SEREs, and used as operands of temporal operators.

It can be argued whether PSL is sufficiently “human readable”, but it is certainly more expressive than languages such as LTL or CTL. It is also “the industry’s first and foremost standard property specification language”, and widely used in assertion-based verification (Eisner & Fisman 2006). As specifications in PSL Foundation Language (which excludes, e.g., PSL macros and built-in functions) can be compiled down to formulas in pure LTL/CTL, tools for model checking are available.

4. Requirement templates for safety-related I&C systems

On the basis of the previous chapters, we outline here a template collection intended for safety-critical I&C systems and limit ourselves to function block oriented control functions and the needs of formal model checking. The solution needs two parts, firstly a set of concepts and their taxonomies and secondly the actual templates in the form of natural language sentences and other information. These are discussed in the sections below. More details can be found in appendix A.

The most visible part of a template is a natural language statement containing placeholders where application-specific terms or more complex logical/arithmetic expressions can be inserted. However, this is not all. The template may need, for example, an explanation and a formal interpretation. So, a template should contain the following information:

- short but descriptive identifier
- long title
- natural language sentence (possibly several alternatives)
- explanation and guidance for use (in text and/or graphics)
- a formal and unambiguous interpretation (possibly several alternative formats)
- links to related templates

In the following, our examples are based on the popular open source model checker NuSMV (<http://nusmv.fbk.eu/>).

4.1 Vocabulary

To be useful, the templates should limit the vocabulary to terms that have a unique and agreed meaning. This vocabulary contains both entity classes (e.g. “temperature transmitter”) and their concrete individuals (e.g. “transmitter TT-101”). Classes have associated properties (“accuracy”) and are linked by various relationships, such as “is-subclass-of” and “is-part-of”. We call the set of entity classes (concepts) and their associated properties and relationships a *domain ontology*. Combined with the actual individuals of the application we have something called a *knowledge base*.

The idea here is that the entity classes, properties and relationships are used to define the templates. This is one way to limit the terms that can be used when applying template. The actual requirement statement typically refers to an individual in the knowledge base. Additional constraints can be defined in terms of rules.

We described the overall landscape in chapter 2. A more extensive discussion can be found in the SAREMAN conceptual model (SAREMAN 2013a). In general, the templates needed to describe a complex system, such as a nuclear power plant or an I&C system, would require

a large number of concepts and their relationships. Therefore, we limit ourselves here to the needs encountered in model checking I&C systems, especially their functions.

For simplicity, we consider here only control system applications based on the function block paradigm (Figure 27). A control application consists of *function blocks* that are instances of *function block types* stored in a function block library. Each function block has an external interface of a number of ports. Ports accept (input) or produce (output) signals of various data types, such as Booleans, integers, real numbers or strings. Moreover, function blocks can have internal variables that store their states and configuration parameters. Complex function block types can be defined by aggregating simple ones. In addition, it is useful to define global variables outside the function blocks.

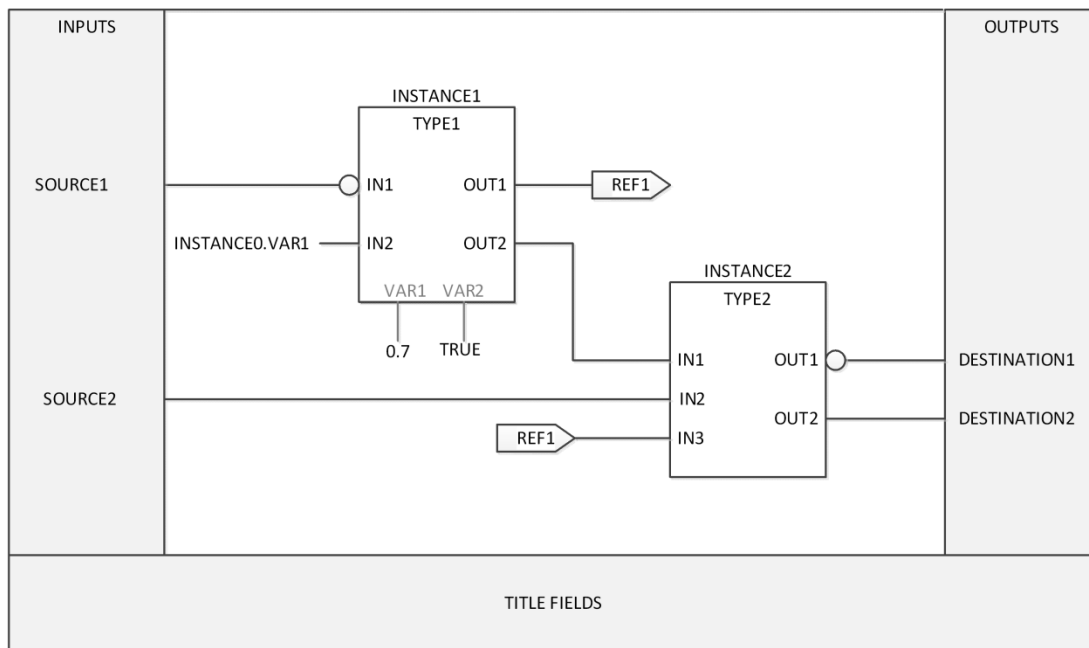


Figure 27. A control application consists of function blocks that are “wired” to each other through their ports. The configuration can be presented graphically on a function block diagram.

The concepts are summarised in Figure 28. A control application is defined by instantiating function blocks from the library, setting suitable values for their configuration parameters and connecting the blocks together through their ports. The application can be presented graphically as *function block diagrams*. For practical reasons, complex applications must be decomposed hierarchically and divided into several diagrams that refer to external variables shown on other diagrams. We expect here that each function block is allocated to *controller* of the I&C system, e.g. to a Programmable Logic Controller (PLC). Blocks are executed in a cyclic manner by a *task* first reading external inputs, then performing the algorithms of each function block in the specified order and finally updating the external outputs.

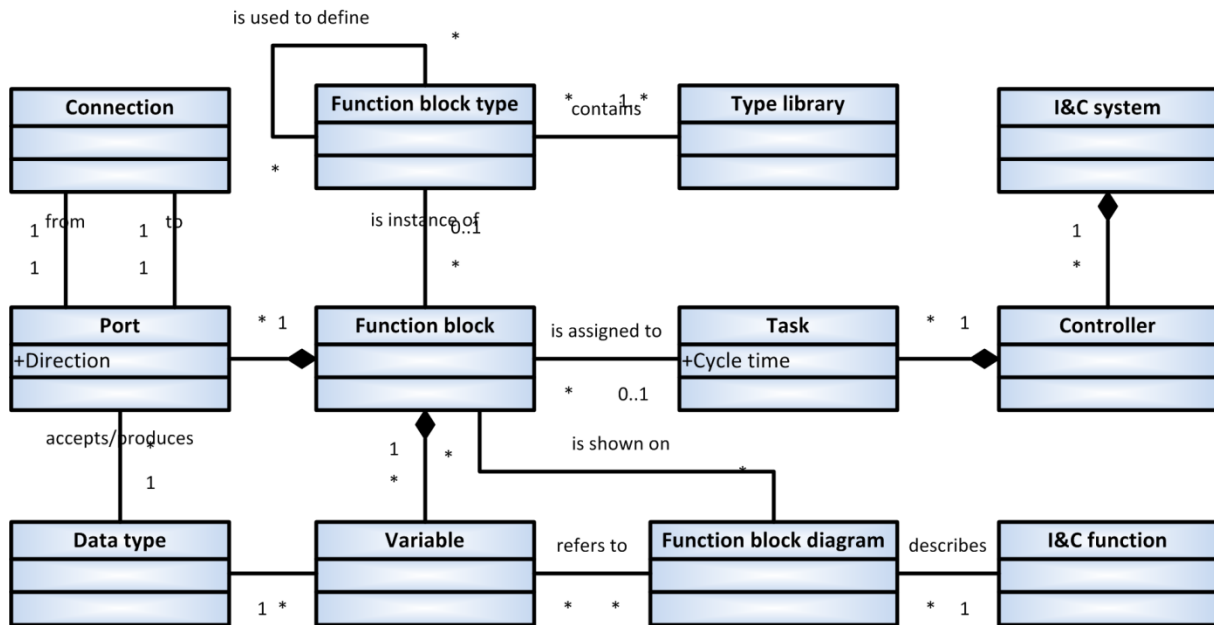


Figure 28. Main elements of a control application.

Note that this abstract domain model applies both to application programs implemented for PLC and distributed control system (DCS) platforms and to their platform-independent functional specifications. Concerning implementations, the programming languages specified by the IEC standard 61131-3 can be used as a point of reference. On the level of functional specifications, function block diagrams can be seen as graphical presentations of *I&C functions*. There is unfortunately no single, widely accepted specification language. Examples of domain-specific suggestions into this direction can be found in (VGB-R 170 C 2004 and NORSOK I-005 2005).

Having now defined the basic elements of a control application, and to make things even simpler, we observe that only variables and ports of function blocks are needed to express the requirements encountered in model checking. It is possible to refer to a variable with its global name or by combining the name of the function block and its port, e.g. "INSTANCE2.IN1" in Figure 27.

For a model checker (section 3.5), both I&C functions and requirements must be transformed into a suitable formal presentation. The same holds also for other computer tools, for example for testing automation (section 3.4). Basically, it would be preferable to express the models and requirements in the terminology used by the designer in the source documents. However, the model checker may have limitations that need to be considered. For instance, data types, timed execution and variable names of function blocks may require attention. The NuSMV input language, for example, has several reserved keywords (e.g., MODULE, DEFINE, VAR, process, array, F, G, EX, AX...) that cannot be used in identifiers of function block types of instances, variables etc. The only special characters allowed in an identifier are underscore, dollar sign, hash and dash (`_$#-`), out of which only the underscore can begin an identifier. An exemplar mapping of designer-oriented concepts to the terms of NuSMV is shown in Table 5. Other mappings are possible, but the presentation here follows the modelling practices found useful at VTT.

Table 5. Mapping of control application concepts to the input language of the NuSMV model checker.

Concept	Presented in NuSMV as
Function block type	MODULE declaration
Function block instance	VAR declaration
Port of a function block	<p>An input variable is introduced as a module parameter, when a MODULE representing the function block is declared. These variables are not explicitly typed, and can be assigned any value in the MODULE code.</p> <p>An output variable is specified with a DEFINE of ASSIGN declaration within the MODULE code.</p> <p>References to these variables are of the type: <i>BLOCK_NAME.VARIABLE_NAME</i></p>
Boolean	boolean
Integer	<p>An enumeration of integer values. A shorthand can be used to define an integer range (e.g., 4 . . 20 for the 4-20 mA signal range). If feasible, the analyst should only allow a minimum number of possible values (that still enables all relevant model behaviours) to limit model state space.</p>
Real	<p>An enumeration of integer values (see above). For sufficient accuracy, the analyst may resort to scaling (e.g., 4203 for 42,03).</p>
Enumeration	enumeration
String	enumeration of string values

4.2 Sentences

Alongside with the restricted vocabulary above, the set of allowed sentence structures, i.e. the CNL grammar, is the second cornerstone of our approach. It was already discussed on a general level in section 2.6. Here we focus on statements encountered in model checking and therefore limit ourselves to a subset of functional requirements.

In model checking, temporal logic is used to describe the required behaviour of a system in terms of states and events occurring during a system's execution. States and events can be expressed as propositions that refer to variables of the control application.

Model checkers use special keywords that in some cases have less obvious interpretations of common words. Our goal is, however, that requirements can be expressed in a way understandable to a control engineer not familiar with temporal logic. Therefore, we need to collect suitable sentence structures and then map them to corresponding expressions in temporal logic. The mappings are described in detail in appendix A. Below we outline the basic elements of natural language sentences typically needed in model checking.

4.2.1 Propositions of states and events

Basic elements of expressions include literals, enumerations, variables and function block ports. A port reference can be optionally written in the form "port <port name> of block <block id>". Simple propositions are formed by comparing an expression to another expression. More complex expressions include simple mathematics and logical combinations.

Let's consider the following example: Figure 29 shows an I&C function with three function blocks: a greater-than comparison block (GT), a set-reset flip-flop (SR), and an on-delay

timer (TON). The idea is that if the level measurement exceeds the value 85, the “flush” command is given after a 10 second delay. Only after the level measurement is below 85, the stored flush command can be manually reset.

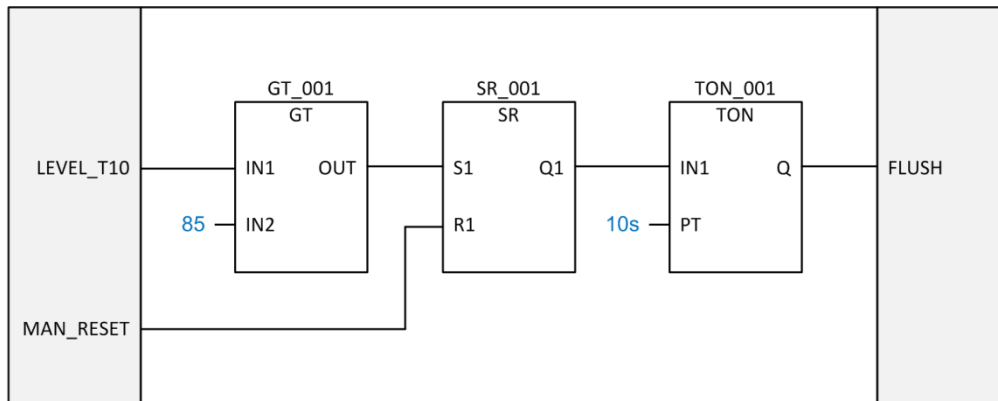


Figure 29. An exemplar I&C function with three function blocks.

The examples below illustrate the basic model elements needed in model checking:

- References for diagram inputs and outputs (LEVEL_T10, MAN_RESET, FLUSH)
- Information on variable data types. Type errors (e.g., LEVEL_T10 = TRUE) can then be avoided.
- Information on input ranges. For example, the LEVEL_T10 input can be modelled as an integer range {84..86}, which in this case is sufficient for capturing all relevant model behaviours.³
- References for function block ports. For example, SR_001.Q1 for the flip-flop block output.

These building blocks are then used to construct propositions that specify states and events of interest. The propositions compute to a Boolean value that can change over the execution of the (system) model. The propositions consist of:

- At the simplest, Boolean variables (for example, MAN_RESET is a proposition; “= true” can be omitted).
- Nested logical (not, and, or, xor, implies, iff), relational (>, <, >=, <=), and arithmetic (+, -, *, /) operators, that at the lowest level refer to variables and constants. For example: ((LEVEL_T10 <= 85) & MAN_RESET).

4.2.2 Reserved words and requirement scopes

We suggested some reserved keywords in section 2.6. Table 6 below tells how we use them in model checking. Discussion on terms used for defining the scopes of requirements can be found in appendix A.

The term “cycle” refers to a time-step in the model. Although real-time tools also exist, model checkers such as NuSMV divide the time line into discrete steps or “states”, on which the

³ The range is relevant for requirement formalisation, because if the analyst is not aware of the possible input values, she/he might formulate a requirement stating, e.g.: “While (LEVEL_T10 = 90) is true, FLUSH will eventually be true”. The result will be a false positive, since the proposition (LEVEL_T10 = 90) will always be false.

model variables are updated, analogous to the way function blocks are processed within a PLC. It is up to the modeller to decide what a time-step means in terms of the modelled application.⁴

Table 6. The interpretation of reserved keywords in CNL.

CNL expression	Interpretation
Always	Entire system execution (globally)
Never	“P is never true” is equivalent with “P is always false”.
Eventually	At the same cycle or at some later cycle
Before P	Cycles up to but not including the cycle where P is true, “Before P becomes true”
After P	Cycles beginning with the first cycle where P is true, “After P first becomes true”
Between P and Q	Cycles beginning with a cycle where P is true, and up to but not including a cycle where Q is true.
After P until Q	Cycles beginning with a cycle where P is true, and 1) up to but not including a cycle where Q is true, or 2) continuing infinitely.
While P	Cycles where P is true
When P, Q	A Boolean implication. False if P & !Q, otherwise true. Same as “P leads to Q”.
P leads to Q	A Boolean implication. False if P & !Q, otherwise true. Same as “When P, Q”.
P changes to true	Any cycle where P is true, and P was false at the previous cycle.

4.2.3 Types of complete sentences

By combining the terms and phrases above it is possible to write complete requirements. The sentences often needed in model checking can be divided in two main types (Lamport 1977):

- **Safety** properties state that “something bad never happens”.
- **Liveness** properties state that “something good keeps happening”.

Informal, written requirement specifications tend to focus on liveness properties. Safety properties are often omitted by designers because they are considered self-evident, and traditionally, their verification has been difficult. In formal verification they must, however, be carefully considered.

At the heart, the sentences specify how the truth values of different propositions change over time. The timing aspect is captured using temporal operators that specify rather the order than the exact timing of states and events.

While temporal logic languages allow for the formulation of complex sequences of states and events, such sequences are difficult to capture using a template-based approach. The reason is that slight alterations in the sequence may lead to a noticeably different temporal logic expression. If the idea is to hide the syntax of temporal logic entirely from the user, we

⁴ A natural selection is to assign one cycle to correspond with the scan cycle of the underlying PLC platform. However, it is often necessary to scale down the time windows allocated to delay blocks (for example, if the PLC scan cycle is 50ms, and there is a function block used to set a 60s delay, it would be grossly inefficient to use 1200 cycles to process the delay).

would need a practically infinite amount of different templates and sentence structures to deal with all possible sequence types (not only varying order but also varying degree of overlap between states). The template examples in Appendix A therefore focus on off-occurring constructs, and sequence representation is a topic for further study.

Combining the elements listed above, and using the templates listed in Appendix A, Table 7 below gives some examples of typical requirements verified with model checking. The CNL representation may appear greatly altered from the original requirement, as the analyst has to rephrase the idea using the set of templates available (see section 5.1).

Table 7. Exemplar CNL requirements for model checking.

Informal requirement	CNL expression
Liveness	
When the tank level exceeds 85 cm, the flush command is given after a 10 s delay.	While condition (MAN_RESET) is false, condition (LEVEL_T10 > 85) is true leads eventually to condition (FLUSH).
The flush command is memorised.	Condition (SR_001.Q1) is true after condition (LEVEL_T10 > 85) is true until condition ((LEVEL_T10 <= 85) & MAN_RESET) is true.
After the tank level returns to allowed limit, the operator is able to manually reset the flush command.	While condition (LEVEL_T10 > 85) is false, condition (MAN_RESET) leads to condition (NOT FLUSH).
Safety	
The flush command is never given without the tank level having exceeded the allowed limit.	Condition (FLUSH) is false before condition (LEVEL_T10 > 85).

5. Working practices and tool support

The previous chapters outlined the principles of a template collection intended for use in the design of I&C systems. Here we discuss what kind of implications the existence of such a collection would have on the designers work and what sort of tool support would be needed. Due to the limited resources, we focus on model checking safety-critical PLC/DCS applications based on the function block paradigm. However, the needs of systems engineer writing more abstract requirements are also considered to some extent.

5.1 Requirements definition in model checking – a task description

When formalising requirements for a model checker, it is reasonable to presume that the task can only begin after the system model (or, at least, the first draft) has been specified. Since the temporal logic constructs must refer to exact and unique model variables, it is necessary – when analysing function block diagrams, in particular – to first select (invoke a type) and name each diagram input, output, or function block object, before any reference to that object can be stated in terms of a formal requirement.

So, we will begin by assuming that a system design has been specified, and the corresponding system model produced by either direct transformation, modelling by hand, or some intermediate form of the two (Pakonen et al. 2013).

Requirement formalisation can begin on the basis of existing requirement documentation, or functional description. There may be an (informal) natural language document containing elements of suitable requirements. Discussions with designers or other system experts may be needed to clarify stated requirements – fill in missing information, and state each

requirement explicitly in terms of system I/O signals or other model variables (examples of such clarifications in the context of writing CNL requirements can be found in Table 7 above).

When using templates for requirement specification, the first task is to find a suitable template. The analyst may already be well aware of which template to select, or some work may be required to find one. Search can be based on template categories, or looking for suitable keywords in template descriptions and metadata.

When a suitable template is found, it is invoked by specifying suitable propositions in terms of model variables.

As a simple example (using the templates from (Dwyer et al. 1998)): The analyst will formalise a requirement that states:

After the temperature in furnace BF-102 exceeds 1300 °C, a horn shall sound in the control room until an operator presses the acknowledge button.

Browsing different template categories, the analyst selects the class “Occurrence”, under which the template “Universality” (a.k.a. “Always”, or “P is true”) seems to fit the general idea. For this template, a scope must be selected. From the requirement, it is quite clear that “After Q until R” is the choice.

The analyst will then specify the propositions P, Q, and R. P stands for “horn sounds”, for which a suitable control output (model variable) is found.

```
P = CR001.Alarms.Horn1.BO1
```

Q stands for “temperature in furnace BF-102 exceeds 1300 °C”, while R stands for “operator presses the acknowledge button”. Again, the analyst specifies the propositions in terms of exact model variables.

```
Q = (DCS002.BF102.Temp200 > 1300)
R = CR001.C_Desk2.ACK_BFTemp
```

The corresponding temporal logic clause for verification will then read:

```
G((DCS002.BF102.Temp200 > 1300) &
!CR001.C_Desk2.ACK_BFTemp -> (G(CR001.Alarms.Horn1.BO1) |
(CR001.Alarms.Horn1.BO1 U CR001.C_Desk2.ACK_BFTemp)));
```

Sometimes, in order to easily state a requirement, it may be necessary to modify the systems model to introduce auxiliary model variables and code. In the PSL standard (IEC/IEEE 2012), the term “satellite” is used of such additions. Alternatively, such modifications may not be strictly necessary, but more convenient and easily readable than the corresponding temporal-logic-only option.

Let us consider two examples. First, if it is necessary to state a requirement that deals with a specific time interval, for example:

After the REQUEST command has been set for 25 cycles, the RESPOND output shall be set.

If PSL is supported by the model checker, we can easily specify:

```
assert always (REQUEST -> next[25] (RESPONSE));
```

Were PSL not supported, the modeller will either have to resort to a very convoluted LTL formula, or define a satellite. In this case, it is quite straightforward to specify a simple and

reusable timer module that counts the cycles over which an input has been set (using the NuSMV input language):

```

VAR
    REQUEST_SET : counter(REQUEST);

MODULE counter(enable)
VAR
    clock : 0..200;
ASSIGN
    init(clock) := 0;
    next(clock) :=
        case
            !enable: 0;
            clock > 199 : 200;
            enable: clock + 1;
            TRUE: 0;
        esac;
  
```

Now the requirement can easily be stated using LTL:

```
G(( REQUEST & (REQUEST_SET.clock = 24) ) -> RESPONSE);
```

Another example has to do with voting between many criteria. Let us say that there are five different alarm criteria (P_1 to P_5), each specified with rather complex propositions. If we now want to state a requirement of the type:

If more than 2 of the alarm criteria P_1 to P_5 are true, the ALARM signal shall be set.

The input language of the NuSMV model checker allows us to formulate the requirement rather straightforwardly using LTL alone, although the resulting formulation might become quite repetitive and complicated (see the “When more than n” pattern in appendix A). If this is not convenient, additional model variables can be introduced to count the number of criteria:

```

DEFINE
    P1_i := [logic that specifies value of P1] ? 0 : 1;
    P2_i := [logic that specifies value of P2] ? 0 : 1;
    P3_i := [logic that specifies value of P3] ? 0 : 1;
    P4_i := [logic that specifies value of P4] ? 0 : 1;
    P5_i := [logic that specifies value of P5] ? 0 : 1;
    ALARM_CRITERIA := P1_i + P2_i + P3_i + P4_i + P5_i;
  
```

Now the appropriate LTL formulation is very intuitive:

```
G(( ALARM_CRITERIA > 2) -> ALARM);
```

Especially if different requirements are based on the same criteria, the analyst may find the use of satellite code more convenient, even if the corresponding logic were possible to state resorting only to temporal logic.

When a model checker returns a counter-example demonstrating a system model behaviour that is contrary to a specified requirement, the first task for the analyst is to investigate if the counter-example is due to either an error in the way the requirement is formalised, or an error in the way the system model represents the actual design. It is often the case that the requirement formalisation for example fails to take into account some exception (for example, the requirement not holding due to an active safety or interlock signal, or a manual operator or maintenance action). To fix the error the analyst will then review the formalisation, and

either make small adjustments to the propositions, or select and altogether different template and start from scratch.

It may even be the case that analyst did not take the exception into account when first formalising the requirement because the fact was omitted from the source documentation (requirement document or functional description). It may become necessary to refine earlier documents.

5.2 Requirements of a template-based requirement editor

We assume that the *analyst* (who can also be the same person as the *designer*) works with a toolset that integrates the following main elements (Figure 30):

- *Domain ontology*: Terms, concepts and their relationships in the application domain, in this case nuclear power plant I&C systems. The ontology may consist of several parts (e.g. upper-level concepts and sub-ontologies for various system types) that are combined according to the needs of the application.
- *Type library*: Definitions of reusable components, such as function blocks and device types. It may actually consist of several parts all referring to the domain ontology.
- *Model editor*: Dedicated tool used by the analyst to build a formalized system model needed by the model checker (Pakonen et al. 2013).
- *Design database*: Requirements, functions and physical system elements of the particular application. Objects in the system model are instances of the classes defined in the domain ontology and the types in the type library.
- *Design tool*: Tool(s) normally used for specifying the I&C system, e.g. a function block editor.
- *Template collection*: Defines the allowed terms and sentence structures with their corresponding formalized counterparts. The template collection uses the vocabulary defined in the domain ontology.
- *Requirement editor*: This is the tool that the analyst uses to write natural language requirements and to generate their formalized versions for the model checker.
- *Model checker*: Software tool used for model checking on the basis of the formalized system model and requirements.
- *Analyst*: Person using the requirements editor and the model checker.
- *Designer*: Person carrying out normal design tasks and using the design tools for that.

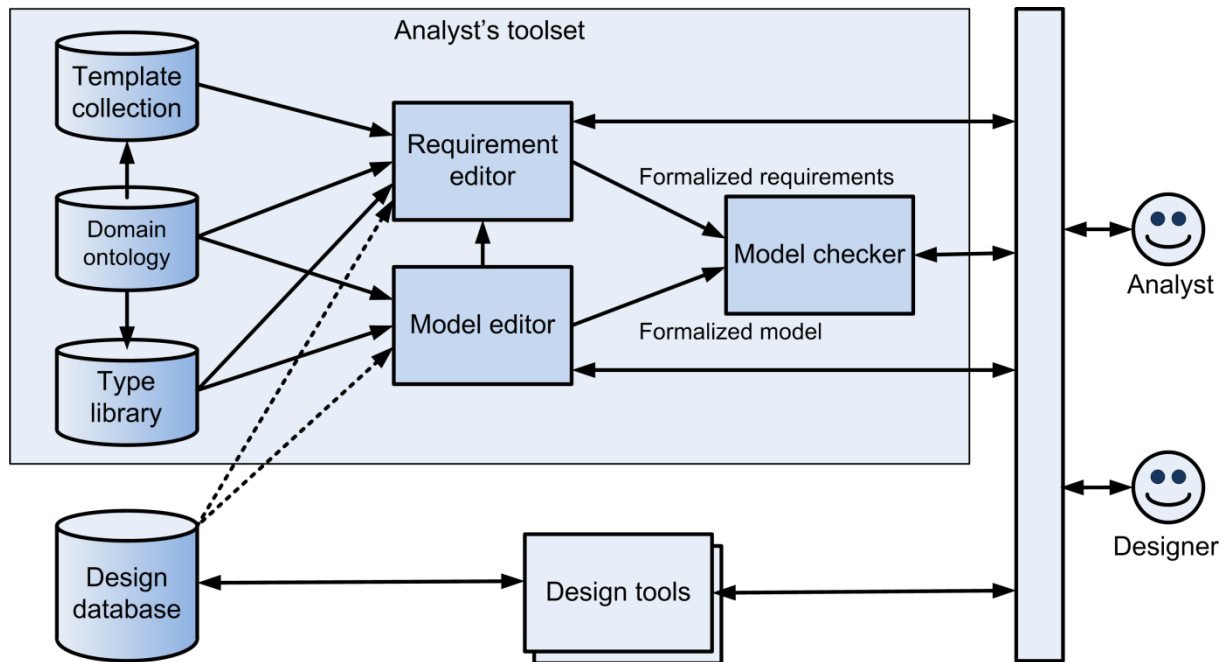


Figure 30. Elements of the design and analysis toolset.

With this abstract “system concept” and the task description above we are able to describe the desired features of a requirements editor. At this stage, we are looking for a tool concept rather than a practical implementation. Therefore, the requirements below are presented informally. We also focus on model checking and written requirements with the length of one sentence. More complex requirements would need other presentation methods, such as diagrams and structured use cases.

Information content

- The kinds of terms that can be inserted into the sentence include
 - fixed terms defined by the template, e.g. keywords like “while”, “when”, etc.
 - literal truth values (true, false), numbers (5, -10.9), enumerations and strings
 - verbs and property names defined in the domain ontology
 - names of individual elements in the system model, e.g. variables and functions
- A (model checking) project in the requirements editor is configured to use certain ontologies and template collections. It is possible to add project-specific concepts to the ontology.
- Propositions can include basic Boolean (and, or), numeric (comparison, addition, multiplication) and string operations (comparison, concatenation, enumeration). In addition, it should be possible to use functions (e.g. abs, sqrt) if supported by the model checker (or some other tool).
- The template collection and the requirement editor are not tied to any particular domain or formalism, such as temporal logic, but can be easily configured for other purposes, for example to be used in automated testing of I&C systems.
- There should be a possibility to select between more than one output formats, for example LTL, CTL or PSL.

- In addition to the requirement text, a structured requirement has attributes like status, date and comment. Some of the metadata, e.g. the type of requirement, can be derived automatically from the templates.
- Template collections are subject to version control and tracking. Once tested and accepted, template collections can be frozen so that they can't be modified accidentally.
- However, if templates (or ontology) must be modified, the way requirements are expressed can be done by changing, e.g. a term, in just one place (from Dick & Llorens 2012)
- Since the editor is used in safety-critical domains, it doesn't allow errors or ambiguities in the templates or in the written requirements. For example, a requirement typed by the analyst must match with exactly one template to be accepted and formalised. However, uncompleted requirements can be saved and corrected later. Error messages and warnings are logged so that the final, error-free log can be used as evidence.
- In general, it should be possible to specify that a part of a sentence, e.g. a property value, is not yet decided so that it can be defined later. This option can also be used in cases where the values are confidential (from Johannessen 2012).

User interface

- The analyst is able to browse the domain ontology, type library and the system model and view all relevant information in it, for example definitions of terms and function blocks and the decomposition of system elements and functions. The user interface is similar to the one normally used in engineering tasks.
- The analyst is able to search and browse the template collection(s) configured to the current project, for example in order to identify the templates suitable for re-formulating a particular unstructured requirement. To make this possible, the templates should have descriptive names and associated keywords. The contents of the templates can be viewed easily.
- Templates are presented in a form that is easily understandable. For better readability, the templates should allow optional clarifying words. For example, when a state is meant, the word "condition" can be added for clarity. The collection provides explanations and guidance for each template.
- The analyst has two options in writing a requirement. She/he can first select a template and fill-in missing parts, or just start typing guided by the editor. In both modes the editor should, for example, suggest suitable next terms and inform about matching templates and kinds of propositions.
- Once the analyst has selected the template she/he wants to use, the editor provides guidance and automation for filling-in the missing parts of the structured sentence. Terms can be typed or dragged and dropped from the system model or a list of possible next terms in the ontology.
- The requirement editor clearly indicates the terms that can be inserted but at the same time prevents the analyst from writing nonsense requirements that violate the rules of the domain ontology, the type library or the system model. Errors are indicated and explained.
- If the analyst notices that a wrong template was selected, she/he is able to transfer the sentence or parts of it (e.g. propositions) to another template.

- It should be possible to write a sentence that doesn't match to any template. For systems engineering in general, the necessity of conforming exactly to a template and domain ontology should be relaxed, e.g., by using NLP techniques (from Arora et al. 2013). Especially in model checking, there shall be a way to write a formalized requirement (e.g. in temporal logic) "manually" in cases for which no template is available in the template collection.
- The user can search the requirements that use a template or an ontology term (from Dick & Llorens 2012)
- The requirement editor should maintain the status of each requirement, that is, whether the model checker proved the requirement true or false the last time it was run. The editor should also state if the status is up to date, that is, have there been any changes to the requirement or the system model after the last time the model checker was run.
- The structured natural language requirements created with the editor can be printed to a human-readable document.

External interfaces

- Requirement information can be imported from and exported to other tools like normal office tools and Requirements Management systems.
- It is possible to trace a structured requirement to the unstructured requirements and other engineering data (or standards) behind it.
- The design database created by the design tools can be used as the system model, for example by importing relevant data or by providing a direct interface to the database.
- It is possible to use existing vocabularies and ontologies as a basis for the domain ontology.
- The formalized requirements (in temporal logic) generated by the tool can be saved (to a file) and exported to other tools, in particular to the model checker.

We can use the experimental CNL editor developed earlier by VTT to illustrate some features of ontology and template based requirements authoring (Tommila, Pakonen & Valkonen 2013). Figure 31 shows the exemplar control application and requirements in section 4.2. The tree in the leftmost frame contains the classes of the domain ontology and the individual function blocks and variables of the flush application. The frame in the middle shows a list of the requirements already inserted to the database. To add a new requirement, the analyst can build a sentence by dragging and dropping allowed next terms from the frame in the lower right corner. If she/he decides to use the keyboard, the editor provides for autocompleting unfinished words. So, this prototype doesn't force the user to first select a template, but is rather intended to be a writing assistant. Therefore, it guides the user by showing a list of allowed next terms and their types, as well the templates matching the text typed in so far. In addition to the natural language templates, there are templates that help in writing temporal logic expressions directly. When the requirement is completed and matches to exactly one template, the editor is able to generate its formal presentation in temporal logic. For example, the sentence "Condition (FLUSH) is false before condition (LEVEL_T10 > 85)" matches to the Absence before template in Appendix A and is transformed into the temporal logic formula $LTLSPEC F(LEVEL_T10 > 85) \rightarrow (! (FLUSH) \cup (LEVEL_T10 > 85))$.

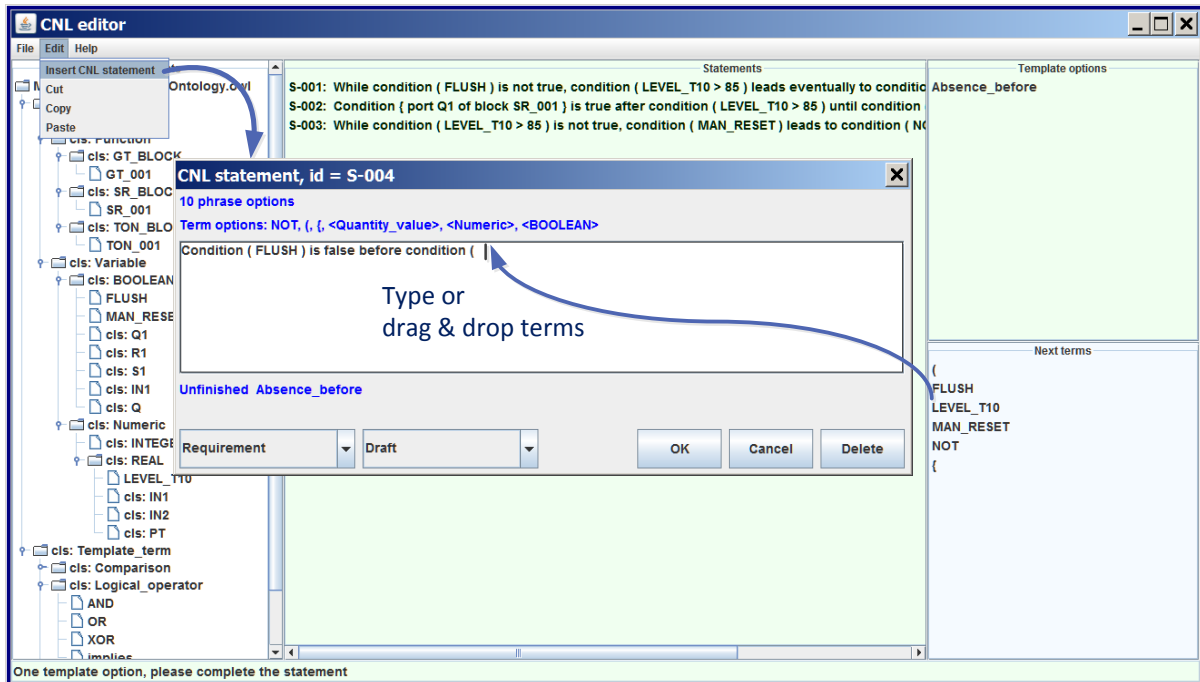


Figure 31. An experimental tool guides the user in writing correct requirements on the basis of a domain ontology, elements of the particular application and a collection of 52 templates.

6. Summary and conclusions

In this research report we have studied the possibilities of Controlled Natural Languages (CNL) in improving the quality of control system requirements. Their limited vocabularies and sentence structures guide designers in writing well-formed requirements that can also be analysed and formalized with computer tools.

Our particular application area is safety-critical I&C systems in nuclear power plants. The wider context is requirements definition in systems engineering in general. Within this area, we have focused on verification of I&C functions with formal model checking where practical solutions and software tools can be foreseen.

On the basis of the literature and our own experiences in model checking, we have collected a first set of functional requirement templates for function block based control applications. In this approach, templates define pairs of allowed sentence structures and corresponding formal interpretations in temporal logic. In our tool concept, actual requirements stated in terms of the variables of the control system can be automatically transformed into a temporal logic presentation and imported to a model checker for verification.

As is widely known, requirements definition and management continue to be a challenge in systems engineering. Also in model checking, there is a real need to make the modelling of the control system and its requirements easier. Our literature review shows that this problem has been recognised by many others, as well. There exists a multitude of techniques and software tools for natural language processing and system modelling. Even quite similar template-based tools have been suggested. In other words, we seem to have a need and ingredients for its solution.

However, this research report represents a first step only. A question to be answered is, how the available ingredients should be combined and adapted to the needs of safety-critical I&C. Concerning systems engineering in general, we believe that guided requirements authoring combined with model-based design and standards-based tool integration is a promising direction in spite of the challenges.

In our particular interest area, model checking, the template-based approach seems to be implementable and solve many practical problems. However, there are also several issues to be resolved. Firstly, single natural language sentences work well only in simple cases. Complex chains of events pose a challenge in the domain of I&C systems. Due to the dynamic behaviour of the physical processes being controlled, system requirements often have to be specified in terms of sequences of varying complexity. Writing down such sequences in CNL results in convoluted sentences. It would be more effective and user-friendly to use structured scenarios or graphical representations, such as sequence diagrams. Furthermore, if the idea is that templates hide the formal syntax of temporal logic languages, such as LTL or CTL, we would need a practically infinite amount of templates to represent all possible sequences.

Secondly, the need for exact and explicit formalisation means that the semantics of reserved keywords in our CNL should be defined in great detail but in way that is familiar to control engineers. Properly understanding the specific meaning of terms like “while”, “when”, “after”, and “before” may require some study from the user. What “after”, for example, actually means in terms of model states and time cycles may be different from the common sense “feel” for the word. The domain ontology, keywords and sentence structures proposed in this report represent a first draft, and practical experimentation and refinements are still needed.

For help with complex sequences, Property Specification Language seems to hold promise. Syntactic sugar helps in understanding the temporal operators, but the real power is in the Sequential Extended Regular Expressions, or SEREs. Similar in spirit to standard regular expressions, SEREs provide a way to describe multi-cycle behaviour that is much more user-friendly than anything that can be expressed with LTL or CTL. Potential new topics for future research include visualisation of SERE constructs, and templates based on SEREs. Furthermore, requirements authoring tools need to be integrated to other tools used in model checking.

Even if further research is needed, practical work at VTT has already shown that requirement templates are very useful in model checking. The majority of functional I&C requirements (especially in safety-critical systems) can be expressed with three types of statements: 1) State A is always true; 2) State B is never true; and 3) State C leads eventually to state D. In addition, a fairly limited collection of domain-specific templates will go a long way further. Although there remains a need to work with temporal logic operators to capture some event chains, it is evident that a requirement specification tool would certainly benefit from templates.

From the systems engineering point of view, the requirement authoring tool concept should be based on a system model that is shared with other modelling, development, analysis, and requirement management tools. The problem with popular requirement management tools is that they are basically word processors handling free text with little or no built-in support for system models and good sentence structures. Fortunately, different extension mechanisms can be used. Also relevant is the integration with modelling and development tools, as the requirements must be anchored to system model elements. Various integration techniques exist but domain-specific classifications should also be agreed to be able to attach meaningful metadata attributes to the requirements.

References

- Allen, J. 1984. Towards a General Theory of Action and Time. *Artificial Intelligence* 23 (1984): 123–54.
- Armengaud et al. 2012. Integrated tool-chain for improving traceability during the development of automotive systems. *ERTS 2012 – Embedded Real Time Software and Systems*, 10 p.
- Arora, C. et al. 2013a. Automatic Checking of Conformance to Requirement Boilerplates via Text Chunking: An Industrial Case Study. *ESEM'13*, 2013, 10 p.
- Arora, C. et al. 2013b. RUBRIC: A Flexible Tool for Automated Checking of Conformance to Requirement Boilerplates. *ESEC/FSE'13*, August 18–26, 2013, Saint Petersburg, 4 p.
- Bacherler, B, Moszkowski, B., Facchi, C. & Huebner, A. 2012. Automated Test Code Generation Based on Formalized Natural Language Business Rules. *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, 7 p.
- Beatty, J., Ferrari, R., Vijayan, B., Godugula, S. 2011. Seilevel's Evaluations of Requirements Management Tools: Summaries and Scores. *Seilevel White Paper*, 2011, 12p.
- Buschmann, F., Henney, K. & Schmidt D. 2007. *Pattern-oriented software architecture – On patterns and pattern languages*. Chichester, John Wiley & Sons Ltd, 450 p.
- Carvalho, G. et al. 2013. Test Case Generation from Natural Language Requirements based on SCR Specifications. *SAC'13 March 18-22, 2013, Coimbra, Portugal*, 6 p.
- CESAR 2010. Definition and exemplification of RSL and RMM. Publication D_SP2_R2.1_M1 of the CESAR project, <http://www.cesarproject.eu/>, 188 p.
- CESAR 2011. Revised Definitions of Improved RE Methods. Publication D SP2 R3.3 M3 of the CESAR project, <http://www.cesarproject.eu/>, 148 p.
- Clark, E. M., Grumberg, O., Peled, D. 1999. *Model Checking*. The MIT Press, 1999.
- Crilly, N. 2010. The roles that artefacts play: technical, social and aesthetic functions. *Design Studies*, 31(4), 311–344.
- Dick, J. & Llorens, J. 2012. Using statement-level templates to improve the quality of requirements. *Integrate systems engineering ltd*, available at http://www.integrate.biz/downloads/White_Paper-templating.pdf.
- Dwyer, M. B., Avrunin, G. S., Corbett, J. C. 1998. 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.
- Dwyer, M. B., Avrunin, G. S., Corbett, J. C. 1999. Patterns in Property Specifications for Finite-State Verification. *Proceedings of the 21st International Conference on Software engineering (ICSE '99)*, ACM, New York, 1999.
- Eisner, C., & Fisman, D. 2006. *A Practical Introduction to PSL*. Springer Science. 240 p.

- Esser, M. & Struss, P. 2007. Automated Test Generation from Models Based on Functional Software Specifications. In: Proc. 3rd Indian International Conference on Artificial Intelligence (IICAI-07), December 17-19 2007, 14 p.
- Farfeleder et al. 2011. DODT: Increasing Requirements Formalism using Domain Ontologies for Improved Embedded Systems Development.
- Fuchs, N., Kaljurand, K. & Kuhn, T. 2008. Attempto Controlled English for Knowledge Representation. In: Baroglio, C. et al. (Eds.): Reasoning Web 2008, LNCS 5224, pp. 104–124, 2008. Springer-Verlag Berlin, Heidelberg 2008.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 395 p.
- Garbacz, P. 2006, Towards a standard taxonomy of artifact functions. Applied Ontology, 1 (3-4): 221-236.
- Hull, E., Jackson, K. & Dick, J. 2002. Requirements engineering. London, Springer-Verlag, 213 p.
- Hametner, R., Winkler, D. & Zoitl, A. 2012. Agile Testing Concepts Based on Keyword-driven Testing for Industrial Automation Systems. IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society, 6 p.
- Hirtz, J., Stone, R., McAdams, D., Szykman, S. & Wood, K. 2002. A Functional Basis for Engineering Design: Reconciling and Evolving Previous Efforts. Research in Engineering Design 13 (2002) 65–82.
- IEC 62381 2011. Automation systems in the process industry - Factory acceptance test (FAT), site acceptance test (SAT), and site integration test (SIT). International Electrotechnical Commission, 40 p.
- IEC/IEEE 2012. IEC 62531:2012, IEEE Std. 1850-2010, Property Specification Language (PSL). International standard, June 2012, 177p.
- IEC 81346-2 Ed.1: Industrial systems, installations and equipment and industrial products – Structuring principles and reference designations – Part 2: Classification of objects and codes for classes. Final draft international standard, 43 p.
- ISO/IEC 29148 2011. Systems and software engineering — Life cycle processes — Requirements engineering. Final draft international standard, March 2011, 94 p.
- Johannessen, V. 2012. CESAR - text vs. boilerplates - What is more efficient - requirements written as free text or using boilerplates (templates)? Norwegian University of Science and Technology, Master's thesis in Computer Science, 88 p.
- Kaljurand, K. & Kuhn, T. 2013. A Multilingual SemanticWiki Based on Attempto Controlled English and Grammatical Framework. In Proceedings of the 10th Extended Semantic Web Conference (ESWC 2013), Springer, 2013. Available at: <http://attempto.ifi.uzh.ch/site/pubs/>.
- Kuhn, T. 2013. A Survey and Classification of Controlled Natural Languages. To appear in Computational Linguistics, 50 p. Available at: <http://attempto.ifi.uzh.ch/site/pubs/>.
- Lamport, L. 1977. "Proving the correctness of multiprocess programs", IEEE Transactions on Software Engineering. Vol 3, Issue 2, pp. 125-143, 1977.

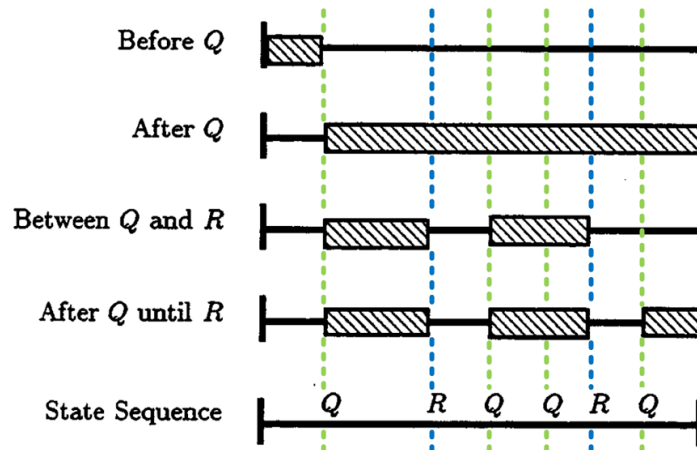
- Lind, M. 2005. Modeling Goals and Functions of Control and Safety Systems - theoretical foundations and extensions of MFM. Electronic report NKS-114, 45 p.
- Mavin, A., Wilkinson, P., Harwood, A. and Novak, M. 2009. Easy approach to requirements syntax (EARS). In: Proceedings of 17th IEEE International Requirements Engineering Conference (RE'09), pp. 317-322, 2009.
- Mavin, A. and Wilkinson, P. 2010. Big ears (the return of easy approach to requirements engineering). In: Proceedings of the 18th IEEE International Requirements Engineering Conference (RE'10), pp. 277-282, 2010.
- NORSOK I-005 2005. System control diagram. Standards Norway, NORSOK standard I-005, rev. 2, April 2005.
- OMG 2013. Requirements Interchange Format (ReqIF). Version 1.1. Object Management Group 2013. <http://www.omg.org/spec/ReqIF/1.1>.
- Pakonen, A., Mätäsniemi, T., Lahtinen, J., Karhela, T. 2013. A Toolset for Model Checking of PLC Software, 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2013), Cagliari, Italy, 10th-13th September, 2013, 6p.
- Reuse 2013. Requirements Quality Analyzer for DOORS, Requirements Authoring Tool, User Guide. The Reuse company, www.reusecompany.com/technical-documentation-rat.
- SAREMAN 2013a. Conceptual model for safety requirements specification and management in nuclear power plants. Working report of the SAREMAN project, version 2, 91 p.
- SAREMAN 2013b. A control engineer's introduction to requirements engineering. Working report of the SAREMAN project, 37 p.
- Schnelte, M. 2009. Generating Test Cases for Timed Systems from Controlled Natural Language Specifications. Third IEEE International Conference on Secure Software Integration and Reliability Improvement, 6 p.
- Solís, C. & Wang, X. 2011. A Study of the Characteristics of Behaviour Driven Development. 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications, 5 p.
- Sowa, J. F. 2011. Introduction to Common Logic. Presentations slides, 38 p, www.jfsowa.com/talks/clintro.pdf.
- SPEEDS 2008. Contract Specification Language (CSL). SPEEDS Deliverable: D.2.5.4, 22 p., available: <http://www.speeds.eu.com/>.
- Tommila, T., Pakonen, A. & Valkonen, J. 2013. Ontology-Driven Natural Language Requirement Templates for Model Checking I&C Functions. Enlarged Halden programme group meeting, EHPG-2013, Storefjell, Norway, 10th – 15th March, 2013, 9 p.
- van Renssen, A.S.H.P. 2005. Gellish - A Generic Extensible Ontological Language - Design and Application of a Universal Data Structure. Delft University Press, 268 p.
- VGB-R 170 C 2004. Function-related documentation of power plant instrumentation and control in line with operating requirements.

Appendix A: Model checking templates for safety critical I&C

Common templates from Dwyer et al.

Discussion on scopes

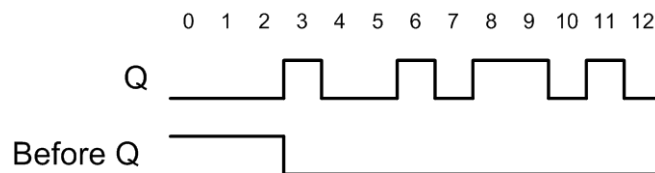
Dwyer et al. use scopes to designate the extent of model execution over which a property must hold. To illustrate the scopes, a figure is used (the dashed lines have been added for clarity):



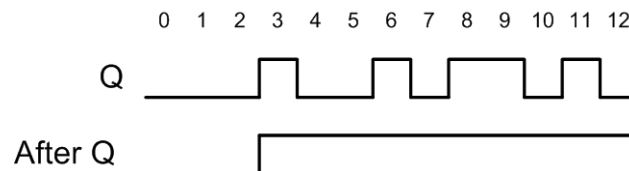
The scopes illustrated (Modified from (Dwyer et al. 1998))

To further clarify the scopes, we will list examples of traces for each scope.

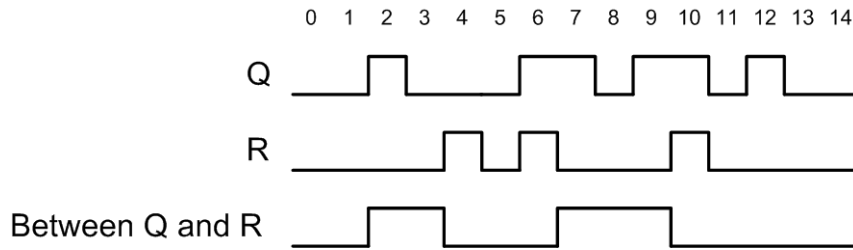
“Before Q” means all cycles up to but not including the first cycle where Q is true. Later values of Q are irrelevant:



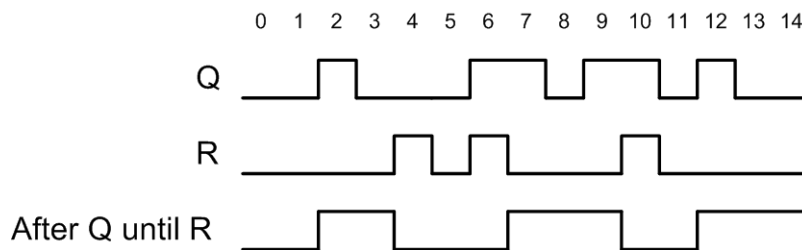
“After Q” means all cycles beginning with the first cycle where Q is true. Note that the “beginning with” –aspect is by no means assumed in the common language use of the preposition “after”. Later values of Q are irrelevant:



“Between Q and R” means all sequences of cycles beginning with the cycle where Q is true, and up to but not including the cycle where R is true. Note that the eventual occurrence of R is needed to initiate the sequence on Q:



“After Q until R” means all sequences of cycles beginning with the cycle where Q is true, and either 1) up to but not including the cycle where R is true, or 2) continuing infinitely in the absence of an occurrence of R. This scope differs from the previous scope in that an eventual occurrence of R is *not* needed to initiate the sequence on Q. A suitable analogy would be a machine with “start” (Q) and “stop” (R) buttons:



Note that in the “Between Q and R” and “After Q until R” scopes, R has “priority” over Q – the occurrence of Q has no effect while R is true. This is not addressed in the scopes illustration used by Dwyer et al., but it can be deduced from the formulas.

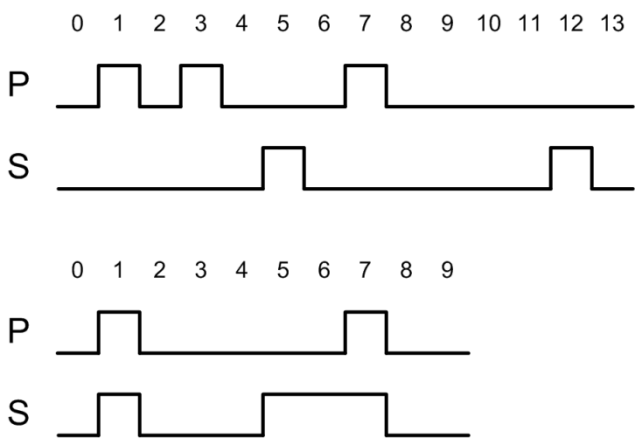
Selected templates

We list here the most common “specification patterns” by Dwyer et al. The rest of the patterns can be found at <http://patterns.projects.cis.ksu.edu/>.

Title:	Universality		
Sentence:	P is true		
Formulas:	LTL	Globally	$G(P)$
		Before R	$F(R) \rightarrow (P \cup R)$
		After Q	$G(Q \rightarrow G(P))$
		Between Q and R	$G((Q \& !R \& F(R)) \rightarrow (P \cup R))$
		After Q until R	$G(Q \& !R \rightarrow (G(P) \mid (P \cup R)))$
	CTL	Globally	$AG(P)$
		Before R	$!E[!R \cup (!P \mid AG(!R)) \& !R]$
		After Q	$AG(Q \rightarrow AG(P))$
		Between Q and R	$AG(Q \& !R \rightarrow !E[!R \cup (!P \mid AG(!R)) \& !R])$
		After Q until R	$AG(Q \& !R \rightarrow !E[!R \cup (!P \& !R)])$
	PSL	always (P)	
Guidance:	The template specifies that a proposition P must hold.		

Title:	Universality
	Note the symmetry with the “Absence” pattern. Stating “always !P” is equivalent to stating “never P”.
Keywords:	Always, Henceforth
Related templates:	Absence
Status:	Well-known , published in collections of Dwyer et al.
References:	(Dwyer et al. 1998) (Dwyer et al. 1999)

Title:	Absence			
Sentence:	P is false			
Formulas:	LTL	Globally	$G(!P)$	
		Before R	$F R \rightarrow (!P \cup R)$	
		After Q	$G(Q \rightarrow G(!P))$	
		Between Q and R	$G((Q \& !R \& F(R)) \rightarrow (!P \cup R))$	
		After Q until R	$G(Q \& !R \rightarrow (G(!P) \mid (!P \cup R)))$	
	CTL	Globally	$AG(!P)$	
		Before R	$!E[!R \cup (!(!P \mid AG(!R))) \& !R]$	
		After Q	$AG(Q \rightarrow AG(!P))$	
		Between Q and R	$AG(Q \& !R \rightarrow !E[!R \cup (!(!P \mid AG(!R))) \& !R])$	
		After Q until R	$AG(Q \& !R \rightarrow !E[!R \cup (P \& !R)])$	
	PSL	<code>never (P)</code>		
	Guidance:	<p>The template specifies that a proposition P must not hold.</p> <p>Typically used to state that something “bad” should never happen (safety property). A common example is mutual exclusion (never A & B).</p> <p>Note the symmetry with the “Universality” pattern. Stating “never !P” is equivalent to stating “always P”.</p>		
	Keywords:	Never		
	Related templates:	Universality		
	Status:	Well-known , published in collections of Dwyer et al.		
References:	(Dwyer et al. 1998) (Dwyer et al. 1999)			

Title:	Response			
Sentence:	P leads eventually to S			
Formulas:	LTL	Globally	$G(P \rightarrow F(S))$	
		Before R	$F(R) \rightarrow (P \rightarrow (!R \cup (S \ \& \ !R))) \cup R$	
		After Q	$G(Q \rightarrow G(P \rightarrow F(S)))$	
		Between Q and R	$G((Q \ \& \ !R \ \& \ F(R)) \rightarrow (P \rightarrow (!R \cup (S \ \& \ !R)))) \cup R$	
		After Q until R	$G(Q \rightarrow ((P \rightarrow (!R \cup (S \ \& \ !R))) \cup R) \mid G(P \rightarrow (!R \cup (S \ \& \ !R))))$	
	CTL	Globally	$AG(P \rightarrow AF(S))$	
		Before R	$!E[!R \cup (!((P \rightarrow A[!R \cup (S \ \& \ !R)])) \mid AG(!R)) \ \& \ !R]$	
		After Q	$A[!Q \ W \ (Q \ \& \ AG(P \rightarrow AF(S)))]$	
		Between Q and R	$AG(Q \ \& \ !R \rightarrow A[(P \rightarrow !E[!R \cup (!(!R \cup (S \ \& \ !R)) \mid AG(!R)) \ \& \ !R)])$	
		After Q until R	$AG(Q \ \& \ !R \rightarrow !E[!R \cup (! (P \rightarrow A[!R \cup (S \ \& \ !R)]) \ \& \ !R)])$	
	PSL	always (P -> eventually! (S))		
	Guidance:	<p>The template specifies that an occurrence of an event/state P must be followed by an occurrence of event/state S. It is typically used to state that a request must lead to a response.</p> <p>Note that:</p> <ul style="list-style-type: none"> • A single occurrence of S will satisfy the condition even when there are multiple past occurrences of P. • P and S may occur at the same cycle.  <p><i>Examples of traces where the requirement holds</i></p>		
	Keywords:	Follows, Leads to		

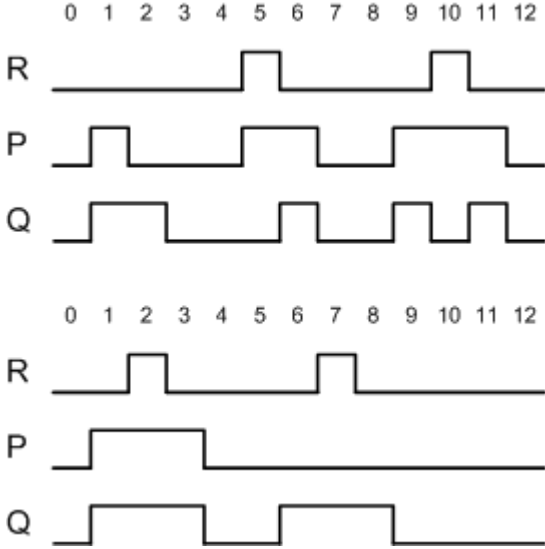
Title:	Response
Related templates:	Conditional response
Status:	Well-known , published in collections of Dwyer et al.
References:	(Dwyer et al. 1998) (Dwyer et al. 1999)

Title:	Existence			
Sentence:	P is eventually true			
Formulas:	LTL	Globally	$F(P)$	
		Before R	$G(\neg R) \mid ((P \ \& \ \neg R) \cup \neg R)$	
		After Q	$G(\neg Q) \mid F(Q \ \& \ F(P))$	
		Between Q and R	$G(Q \ \& \ \neg R \rightarrow (G(\neg R) \mid (\neg R \cup (P \ \& \ \neg R))))$	
		After Q until R	$G(Q \ \& \ \neg R \rightarrow (\neg R \cup (P \ \& \ \neg R)))$	
	CTL	Globally	$AF(P)$	
		Before R	$\neg E[\neg (P \ \& \ \neg R) \cup (R \ \& \ \neg (P \ \& \ \neg R))]$	
		After Q	$\neg E[\neg (Q \ \& \ AF(P)) \cup (Q \ \& \ \neg (Q \ \& \ AF(P)))]$	
		Between Q and R	$AG(Q \ \& \ \neg R \rightarrow \neg E[\neg (P \ \& \ \neg R) \cup (R \ \& \ \neg (P \ \& \ \neg R))])$	
		After Q until R	$AG(Q \ \& \ \neg R \rightarrow A[\neg R \cup (P \ \& \ \neg R)])$	
	PSL	eventually! (P)		
	Guidance:	The template specifies that a proposition P must not hold at some point.		
	Keywords:	Eventually		
	Related templates:			
Status:	Well-known , published in collections of Dwyer et al.			
References:	(Dwyer et al. 1998) (Dwyer et al. 1999)			

Original pattern candidates found useful in VTT customer projects

The templates that follow are based on oft-used requirement constructs that have proven useful in practical work at VTT. The templates serve as examples, and represent work in progress. The collection is by no means considered conclusive.

Title:	Conditional leads to	
Sentence:	While R is not true, P leads to Q	
Formulas:	LTL	$G(\neg R \rightarrow G(P \rightarrow Q));$

Title:	Conditional leads to
	PSL <code>always (!R -> always (P -> Q)) ;</code>
Guidance:	<p>A cause-and-effect relationship of certain model states or signals might not be true at all cycles due to, e.g., an overriding signal with higher priority, or a periodic test sequence or a manual user action temporarily inhibiting normal signal flow.</p> <p>This template allows for the specification of a state or signal R which will (temporarily) break the cause-and-effect relationship between P and Q. That is, if R is not true, P will lead to Q at all cycles.</p> <p>Note that “P leads to Q” (or Q) <i>may</i> still be true regardless of the value of R, it is just that “P leads to Q” <i>has</i> to be true <i>only</i> when R is false.</p> <div style="text-align: center;">  </div> <p style="text-align: center;"><i>Examples of traces where the requirement holds</i></p> <p>Notes on formulas:</p> <ul style="list-style-type: none"> • The Boolean proposition (P -> Q) can be replaced with other logic to achieve other “Conditional” formulas. • The LTL formula G !R -> G (P ->Q); without the surrounding parenthesis means that scenarios where R has been active at some point will not be considered in verification. That is, the verification tool will assume that R can never be true.
Keywords:	conditional, protection, interlock, leads to
Related templates:	Conditional response
Status:	Verified using both a model that is known to fulfil the requirement, and a model that is known not to fulfil the requirement.
References:	

Title:	Conditional response	
Sentence:	While R is not true, P leads eventually to Q	
Formulas:	LTL	$G (!R \rightarrow G (P \rightarrow F Q)) ;$
	PSL	$always (!R \rightarrow always (P \rightarrow eventually! Q)) ;$
Guidance:	<p>A cause-and-effect relationship of certain model states or signals might not be true at all cycles due to, e.g., an overriding signal with higher priority, or a periodic test sequence or a manual user action temporarily inhibiting normal signal flow.</p> <p>This template allows for the specification of a state or signal R which will (temporarily) break the cause-and-effect relationship between P and Q. That is, if R is not true, an occurrence of P must be followed by an occurrence of Q.</p> <p>Note that:</p> <ul style="list-style-type: none"> • “P leads eventually to Q” (or Q) <i>may</i> still be true regardless of the value of R, it is just that “P leads eventually to Q” <i>has</i> to be true <i>only</i> when R is false. • A single occurrence of Q will satisfy the condition even when there are multiple past occurrences of P. • P and Q may occur at the same cycle. 	
Keywords:	conditional, protection, interlock, leads to	
Related templates:	Conditional leads to, Response	
Status:	Verified using both a model that is known to fulfil the requirement, and a model that is known not to fulfil the requirement.	
References:		

Title:	When more than n	
Sentence:	When more than n of the P1, P2, P3, and P4 are true, Q is true	
Formulas:	LTL (smv)	$G ((((P1 ? 1 : 0) + (P2 ? 1 : 0) + (P3 ? 1 : 0) + (P4 ? 1 : 0)) > n) \rightarrow Q) ;$
Guidance:	<p>Voting logic: if more than n (where $0 \leq n \leq 3$) propositions out of the set { P1, P2, P3, P4 } are true, then Q must also be true.</p> <p>Notes on formulas:</p> <ul style="list-style-type: none"> • If less than four of the propositions Pn are needed, simply insert “true” to other places to use the same formula. 	
Keywords:	more than, vote, redundancy	
Related templates:	If less than n	
Status:	Verified using both a model that is known to fulfil the requirement, and a model that is known not to fulfil the requirement.	
References:		

Title:	Trigger	
Sentence:	When P changes to true, Q is true.	
Formulas:	LTL	$G ((!P \ \& \ X \ P) \rightarrow X \ Q) ;$
	PSL	$always ((!P \ \&\& \ next \ P) \rightarrow next \ Q) ;$
Guidance:	The template states that whenever P changes from false to true (i.e., on a rising edge of P), Q must also be true at the same cycle.	
Keywords:		
Related templates:	Delayed trigger	
Status:	Proposed	
References:		

Title:	Delayed trigger	
Sentence:	When P changes to true, Q is eventually true.	
Formulas:	LTL	$G ((!P \ \& \ X \ P) \rightarrow X \ (F \ Q)) ;$
	PSL	$always ((!P \ \&\& \ next \ P) \rightarrow next \ (eventually! \ Q)) ;$
Guidance:	The template states that whenever P changes from false to true (i.e., on a rising edge of P), an occurrence of Q must follow. Note that a single occurrence of Q will satisfy the condition even when there are multiple past occurrences of a rising edge of P.	
Keywords:		
Related templates:	Trigger	
Status:	Proposed	
References:		