Visual formalisms applied to temporal logic property patterns in the nuclear automation domain

Igor Buzhinsky

School of Electrical Engineering

Research report, SAUNA project (SAFIR2018 program)
Espoo 15.12.2016

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT

Author: Igor Buzhinsky

Title: Visual formalisms applied to temporal logic property patterns in the nuclear automation domain

Date: 15.12.2016 Language: English Number of pages: 3+40

Department of Electrical Engineering and Automation

This report is written within the scope of the SAUNA project, a part of SAFIR2018 program. While the general goal of the SAUNA project is to provide structured evidence of nuclear automation systems correctness and safety, this report focuses on user-friendly requirement representation for such systems, in particular in the field of model checking, a popular formal verification technique. Such representation are extremely important since industrial practitioners are often unfamiliar with formal languages, such as temporal logics, which hinders the use of formal methods in industry.

Previous studies examined visual formal specification languages and revealed the most common temporal requirement patterns typical for the nuclear information and control (I&C) industry. Using a case nuclear power plant model, this report illustrates these patterns and also suggests new ones based on specifications formulated for this model. Then, it binds the results related to patterns and visual languages together: it examines whether and how these patterns can be represented in known visual formal specification languages. Based on this examination, several visual languages are indicated to be potentially applicable for user-friendly property formalization in the nuclear I&C domain.

Keywords: visual specification languages, formal verification, model checking, requirement patterns, nuclear power plants

Contents

1 Introduction

This report is written within the scope of the SAUNA project (a part of research program SAFIR2018), whose aims are to provide integrated means of assuring and demonstrating safety of nuclear power plants (NPPs) and their systems. One of the ways to achieve these goals is to use *model checking* [8, 5], a formal verification technique which is able to check properties stated in terms of the state of the system and its change in time. This technique requires that the properties to be checked are formulated in the formal languages of temporal logics such as LTL or CTL. However, formal notations require profound experience to be applied, and the lack of such experience may be dangerous [16] since certain software faults can be missed due to improperly written formal specifications. The lack of appropriate expertise is believed to be among the impediments to the adoption of formal methods [13], and in particular in the Instrumentation and Control (I&C) systems industry. One of subtasks of SAUNA addresses this problem, aiming to examine or create user-friendly means of representing and editing formal specifications.

Previous research [29, 23] has reviewed a number of formalisms, including visual ones, intended to reduce the gap between the complexity of temporal logics and industrial practice. After that, a number of temporal property patterns common for the I&C industry have been identified [22]. This report develops this line of research by examining how well existing visual formalisms can represent the most common temporal property patterns. It also reviews a wider range of patterns than in [22] by utilizing temporal properties specified for a case generic NPP model in the Apros simulation environment. As a result, several visual property specification languages are selected as the ones possible to be applied in the nuclear industry.

The rest of the report is structured as follows. Section 2 reviews previous research on the considered topic. Then, Section 3 introduces the concepts and materials used throughout the report. After that, Section 4 selects and discusses property patterns which are common in the nuclear automation domain. In Section 5, these patterns are expressed using visual specification languages. The findings are analyzed in Section 6, after which Section 7 concludes the report.

2 Related work

While formal methods, such as model checking, are powerful in detecting software faults which cannot be identified by conventional testing [16], these methods have problems in industrial adoption. According to [13], these problems are caused by several factors, including the lack appropriate expertise. Specialists with insufficient experience are in danger of composing erroneous formal specifications and thus failing to reveal software faults. As shown in [14], the familiarity to formal languages takes time and practice to acquire, and basic training is often insufficient. Moreover, even assuming that the specialists have proper training and experience, another problem still remains: the results of verification often must be demonstrated to stakeholders. In the nuclear industry in particular, most stakeholders are not familiar with formal notations. To address these challenges, multiple approaches were proposed, such as introducing a hierarchy of property patterns [12, 13] with distinct semantics, and proposing visual languages such as the TimeLine editor [28, 16].

Much work in reviewing existing user-friendly approaches to formal notations has been done in VTT Technical Research Centre of Finland Ltd. In the report [29], textual representations of formal requirements are studied in the context of nuclear I&C systems. The study examines controlled languages, such as boilerplates and templates, and tools to represent, edit and analyze requirements specified in controlled languages. Complementary to [29], the work [22] shortly evaluates existing visual formalisms in terms of their expressive power, user-friendliness and tool support. As a result, Simulink Design Verifier [20] is named as a visual formalisms which is the most suitable to be used in I&C software verification.

As for the research conducted within the SAUNA project, the work [23] provides a wider review of user-friendly specification languages with less elaboration. In addition to reviewing specification techniques, the difficulties of designing a user-friendly language and the factors which make languages user-friendly are also investigated. The review is concluded with factors which are crucial for a formal specification language to be user-friendly, which are: the proximity of used notations to domain languages used in industry, tool support, and graphical representations.

In [22], a study is performed to reveal the most common temporal operators and patterns in the I&C domain. This study, which is based on temporal properties collected from practical model checking projects in the nuclear industry, shows that the majority of temporal properties used in the nuclear industry can be covered by a relatively small number of property patterns. The weakness of the study is that all the examined properties were written by only two analysts. Nevertheless, the data behind [22] relates to quite a wide range of real world systems with different I&C platforms, vendors and architecture level design.

In this report, the data collected in [22] is augmented with properties specified for a different nuclear system by a different analyst. This system is not that representative of an actual I&C system implementation, but is applicable to illustrate the use of temporal property patterns.

3 Background

3.1 Temporal logics and model checking

Throughout this report, temporal logic [6, 8] formulas will be used widely. In general, these formulas are predicates over Kripke structures, a particular kind of finite-state models. Kripke structures are obtained based on the systems to be verified, and can be represented either explicitly or symbolically.

The formulas of *linear temporal logic* (LTL) are formulated in terms of infinite paths, and the satisfiability of an LTL formula on a Kripke structure is defined as satisfiability of this formula for every valid path in the structure. In addition to Boolean variables (called *atomic propositions*) and operators, LTL formulas may include the following temporal operators:

- **X** ("next"): $\mathbf{X}\, f$ requires that formula f is satisfied for the next state of the path.
- **G** ("globally"): $\mathbf{G}\, f$ requires that f holds for the current state and all further states of the path.
- **F** ("in the future"): $\mathbf{F}\, f$ requires for some further state in the path (or already for the current state) f is satisfied.
- **U** ("until"): $f\, \mathbf{U}\, g$ requires that f holds for a finite number of states, after which g holds for the next state.

A special subclass of LTL properties is *safety properties*. Informally speaking, such properties specify that some undesired condition never happens. Formally, this means that every *counterexample* (i.e. path proving that the property is violated) has a finite prefix which is sufficient to show that the property is violated (i.e. each its continuation to a proper infinite path is a counterexample). In contrast, *liveness properties* assert that some desired condition must happen. Formally, for a liveness property, each execution path of a Kripke structure can be extended to an infinite one so that this property is satisfied. Every other LTL property can be represented as a conjunction of a safety property and a liveness property [2], so it is usually sufficient to consider only these two property classes.

As for *computation tree logic* (CLT), every its temporal operator is annotated with a quantifier **E** ("there exists a path") or **A** ("for all paths"). This allows to specify reachability properties like $\mathbf{EF}\, f$ ("f is true in some reachable state"), but there are LTL properties which cannot be expressed in CTL [5]. Nevertheless, the majority of temporal property patters which will be discussed further in this report can be represented both in LTL and CTL.

The problem of *model checking* refers to checking a temporal formula for a chosen model. In particular, distinct verification algorithms exist for LTL and CTL. If the system to be verified is not a finite-state model (i.e. it cannot be directly converted to a Kripke structure), then the corresponding formal model should be created. For example, this can be done using model checkers such as NuSMV, SPIN and UPPAAL. The process of creating a formal model is often manual (unless the source code of the software to be verified is available) and incorporates model simplification

to reduce the computational complexity of model checking. However, there are means to partially automate this process. For example, in [21], NuSMV models are generated from function block diagrams, and in [7] plant models are generated based on simulation traces and the temporal specification of the plant.

3.2 Visual specification languages

Previous works were used to select visual property specification languages suitable for temporal property representation. The main source of considered visual languages is the work [22]. All eight visual formalisms mentioned in [22] are also considered in this report, namely: graphical interval logic [11], L_R temporal logic [18], timing diagram editor [30], TimeLine editor [28], property sequence charts [4], live sequence charts [9], Simulink design verifier [20], and visual timed event scenarios [1]. Next, the study [23] mentions a larger number of approaches, including some of the ones analyzed in [22]. However, not all of these approaches were found appropriate for the purposes of this report, so only two more visual languages were selected from [23] in addition to [23], namely constraint diagrams [10] and symbolic timing diagrams [25]. In Section 5, the selected visual formalisms will be evaluated with respect to temporal property patterns which will be identified in Section 4.

3.3 Nuclear power plant model

Throughout the report, a generic pressurized water reactor (PWR) NPP model (referred hereafter as "generic PWR model") will be used for the purpose of illustration. This model was provided by Fortum Power and Heat Oy, a power utility with NPP operation license in Finland. It is not intended for public use, so the actual names of its internal parameters will not be revealed in the report.

The generic PWR model is implemented in the Apros Nuclear[1] continuous process simulator provided by Fortum Power and Heat Oy and VTT Technical Research Centre of Finland. Mimicking an existing NPP type, it incorporates the most common components of an NPP, including both process modeling and automation logic. The overview of its components, separated between plant and control subsystems, is shown in Table 1. Each component is represented by one or more Apros process or automation diagrams.

Table 1: Generic PWR model overview.

Process networks	Automation networks
Primary circuit	Reactor control
Pressure vessel	Plant and turbine power control
Emergency system	Reactor and turbine trip
Steam generators	Protection networks
Feedwater systems	Pressurizer pressure and liquid level control
Turbine plants	Steam generator water level control
Etc.	Etc.

[1] `http://www.apros.fi/en/`

The generic PWR model has been previously used in [17]. In this work, the authors describe the structure of the model as follows: "Two main process loops are used for power generation using nuclear energy, the primary and secondary circuit. The primary circuit contains the reactor vessel and the nuclear fission within the fuel generates thermal energy, which heats the water in the vessel. The coolant pumps in the primary circuit circulate water through the steam generators and the reactor vessel and thus thermal energy is transferred from the primary to the secondary circuit. The pressurizer, also part of the primary circuit, is a vessel is partially filled with water and it is designed for pressure regulation using heaters and water sprays. The secondary circuit is connected to the primary through the steam generators. Heat from the primary circuit converts water flowing in the secondary side of the steam generators into steam. Turbines use the high-pressure steam flow and drive electric generators. Condensers are used to convert the low-pressure steam after the turbines back to water".

3.4 Temporal specification for the nuclear power plant model

During the research within the SAUNA project, eight subsystems (mostly protection ones) of the generic PWR model were selected for open-loop and closed-loop model checking using the NuSMV[2] symbolic verifier. To enable model checking, temporal properties were prepared for these subsystems based on the system documentation and the implementation of the Apros model. Some of these properties are explained in Section 4 to illustrate the identified temporal property patterns. While properties meant for open-loop verification specified requirements which must be satisfied for each input sequence to the controller, properties meant for closed-loop verification were only possible to check meaningfully under the presence of the plant model, the behavior of whose sensors is not arbitrary. In total, 265 temporal properties were formulated, although it was common that a single textual requirement corresponded to several temporal properties. Almost all the properties were possible to specify both in LTL and CTL, of which CTL was chosen to shorted model checking time in NuSMV. In Section 4, these properties will be used together with the ones from [22] to identify temporal property patterns common in the nuclear automation domain.

[2] `http://nusmv.fbk.eu/`

4 Temporal property patterns

The idea of temporal property *patterns* was introduced by Dwyer et al. [12, 13] to classify temporal specifications into a number of classes, each having a particular semantics. Based on a sample of specifications, Dwyer et al. developed a hierarchy of LTL properties patterns, such as absence, universality, existence, precedence and response.

In this report, the concept of patterns is also used, although sometimes only one pattern for two or more interpretations will be considered if these interpretations can be represented by similar temporal formulas. More information will be provided below. To gather patterns which are common in nuclear automation industry, the following sources were used:

1. a recent study [22], where 1079 formal properties have been collected from VTT's model checking commissions in the Finnish nuclear industry and then grouped according to identified patterns;

2. 265 temporal properties recently specified for the generic PWR model (see Section 3.4) during the research on the SAUNA project, which were prepared based on both system documentation and implementation.

Here, a "pattern" is an LTL or CTL formula template, where each atomic proposition can be replaced by a custom Boolean formula. There is, however, no need to consider a pattern for each possible Boolean formula, due to the following reasons:

- Some patterns are equivalent, i.e. they are based on equivalent temporal properties, such as $\mathbf{G}(x \wedge \mathbf{X}\, y \rightarrow \mathbf{X}\,\mathbf{X}\, z)$ and $\mathbf{G}(x \rightarrow \mathbf{X}(y \rightarrow \mathbf{X}\, z))$. Also note that equivalence is possible between properties formulated in different temporal logics (e.g. LTL property $\mathbf{G}\, x$ is equivalent to the CTL one $\mathbf{AG}\, x$), but there are properties which cannot be converted into a different temporal logic (e.g. LTL property $\mathbf{F}\,\mathbf{G}\, x$ has no representation in CTL [5]).

- Some patterns can be transformed to each other using simple Boolean transformations, like $\mathbf{G}\, x$ and $\mathbf{G}\, \neg x$, which are connected by the negation operation. For visual formalisms which will be considered in this report, it is common that such transformations do not influence the possibility to represent a certain pattern in a certain formalism and do not alter property representation significantly.

- One pattern can generalize another pattern. This can be achieved either by Boolean transformations ($\mathbf{G}\, x$ is more general than $\mathbf{G}(x \rightarrow y)$, since x can be substituted by $x \rightarrow y$) and by temporal ones ($\mathbf{G}(q \rightarrow \mathbf{G}\,\mathbf{F}\, p)$ is more general than $\mathbf{G}\,\mathbf{F}\, p$ since q can be substituted by `true`, and $\mathbf{G}\,\mathbf{G}$ can be simplified to just $\mathbf{G}$).

To avoid clutter, equivalent patterns and ones that can be transformed into other patterns using only Boolean transformations, are avoided in this report. Since visual property specification languages often do not support direct translation of a temporal formula to this language, translating a property requires understanding its semantics.

Semantics of such properties are identical, so no need in distinguishing them was found. However, the shortcoming of this approach is the inability to compare the frequencies of temporal patterns with other sources, like [12, 13].

Property patterns from [22] included equivalent and redundant ones. Such patterns were merged by selecting one of the most general of them, and summing their frequencies in the data. Moreover, some of the properties (12%) from [22] were specified in *property specification language* (PSL), an extension of linear temporal logic. Almost all non-PSL properties were formulated in LTL (87%) with a tiny fraction of CTL ones. The corresponding PSL patterns from [22] were replaced by more specific ones so that they could be transformed into temporal logic patterns (this extended analysis of PSL patterns was performed based on the original data behind [22]). As for the properties from Section 3.4, all of them were specified in CTL and were directly grouped into distinct patterns. Almost all of these properties are also possible to represent in LTL.

The temporal properties collected in [22] and mentioned in Section 3.4 have one more difference: while the properties from [22] are exclusively of the open-loop type (i.e. checking them does not require the plant to be modeled), the properties from Section 3.4 include both open-loop and closed-loop ones. The difference between these property classes lies in the variables used inside of them (whether they belong to the controller or the plant) but not in the temporal structure, so this separation is not worth detailed attention in this report.

Table 2 lists the patterns identified from both sources. Then, the rest of the section explains each of the patterns in more detail, and illustrates their applicability to the generic PWR model described in Section 3.3. Patterns with high (greater than 5%) abundance in any of the sources (patterns 1–7) will be further used to evaluate visual specification languages in Section 5.

Table 2: Identified temporal logic patterns.

Index	Pattern name	Pattern formula	Frequency in [22]	Frequency in Section 3.4
1	“Always”	$\mathbf{G}\, p$	66.64%	13.58%
2	“Unbounded response”	$\mathbf{G}(p \rightarrow \mathbf{F}\, q)$	2.87%	31.32%
3	“Infinitely often”	$\mathbf{G}\,\mathbf{F}\, p$	0.00%	23.40%
4	“Cannot happen until”	$((\neg q)\, \mathbf{U}\, p) \vee \mathbf{G}\, \neg q$	1.85%	16.98%
5	“Response on the next step”	$\mathbf{G}(p \rightarrow \mathbf{X}\, q)$	6.86%	3.02%
6	“Infinitely often after condition”	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\, q)$	0.00%	9.43%
7	–	always {p[*n]} |-> q!	6.12%	0.00%
8	–	$(\mathbf{G}\, p) \rightarrow (\mathbf{G}\, q)$	2.50%	0.00%
9	–	always {p;q[*n]} |-> r!	1.11%	0.00%
10	–	never {p[*n]}	1.02%	0.00%
11	–	$\mathbf{G}(p \wedge \mathbf{X}\, q \rightarrow \mathbf{X}\,\mathbf{X}\, r)$	0.28%	0.00%
12	“Reachable”	$\mathbf{EF}\, p$	0.00%	1.51%
13	“Eventually”	$\mathbf{F}\, p$	0.00%	0.75%
–	Other	–	10.75%	0.00%

4.1 Pattern "Always"

LTL formula	$\mathbf{G}\, p$
CTL formula	$\mathbf{AG}\, p$
LTL type	safety
Semantics	p is satisfied in all reachable states.
Selected for formalism evaluation?	Yes

The "always" pattern (also known as the "universality" pattern [13]) requires that a certain Boolean condition is an invariant, i.e. it is valid for all reachable states of the system. Unlike [22], in this report universal negative conditions (i.e. the ones of the form $\mathbf{G}\, \neg p$) are in particular regarded as instances of this pattern. The "always" pattern also encompasses the subcase of a universal causal dependency where the premise and the consequence are related to the same time moment ($\mathbf{G}(p \rightarrow q)$).

In the requirements formulated for the considered generic PWR model, the following properties were specified using this pattern:

- The speed of the containment spray pump is always zero (since containment is not included in the generic PWR model, the corresponding emergency pump is not activated).
- Various valves are never opened and closed simultaneously. This is an example of a closed-loop property, but, more precisely, it validates the model of the plant, since it must be satisfied for any possible controller.
- Two output protection signals for a different protection network are always equal (since they are computed based on the same input condition).
- Certain conditions over the pressure in the pressurizer (immediately) lead to open signals of certain valves.
- Certain conditions over the water level in steam generators (immediately) lead to switching the emergency feedwater pumps either on or off.

4.2 Pattern "Unbounded response"

LTL formula	$\mathbf{G}(p \rightarrow \mathbf{F}\, q)$
CTL formula	$\mathbf{AG}(p \rightarrow \mathbf{AF}\, q)$
LTL type	liveness
Semantics	Whenever p happens, q must happen in the future (possibly, on the same time step).
Selected for formalism evaluation?	Yes

The "unbounded response" pattern states that a certain condition over the state of the system must eventually cause another condition. This requirement does not state any timing constraints, thus it can be used when such constraints are not known or not important. The former is a common case for the generic PWR model, which has influenced the frequency of this pattern.

In the requirements formulated for the generic PWR model, the following properties were specified using this pattern:

- If certain dangerous values of key system parameters are reached, then eventually activation signals of emergency pumps shall be generated.
- If steam pressure in either of the steam generators exceeds the mean pressure by a certain value and emergency feedwater lines are not closed, then the corresponding valve closing signals shall eventually be generated.
- If water level in either of the steam generators exceeds the specified value, the voltage is supplied, and the emergency feedwater line is not opened, then the corresponding valve opening signal shall be eventually generated.
- If certain dangerous values of key system parameters are reached, then reactor rod position shall eventually reach zero (i.e. the reactor shall be tripped). This is an example of an closed-loop property.
- If certain dangerous values of key system parameters are reached, then reactor relative power shall be eventually less than the specified value.
- If the pressurizer water level falls below a certain value, then it shall eventually exceed this value (i.e. the water level is always eventually restored).
- If the pressurizer pressure is less/greater than a certain value, then it shall eventually become greater/less than this value.

4.3 Pattern "Infinitely often"

LTL formula	$\mathbf{G}\,\mathbf{F}\,p$
CTL formula	$\mathbf{AG}\,\mathbf{AF}\,p$
LTL type	liveness
Semantics	p is satisfied infinitely often.
Selected for formalism evaluation?	Yes

The "infinitely often" pattern is the special case of the "unbounded response" pattern with the premise being always true, i.e. at every moment, a certain desirable state of the system must occur in the future. The sequence of steps where this condition is fulfilled, thus, becomes infinite. In the specifications for the generic PWR model, however, this requirement is used in one more sense: $\mathbf{G}\,\mathbf{F}\,p$ can be reformulated as $\neg\,\mathbf{F}\,\mathbf{G}\,q$, where $q = \neg p$, which means that infinite sequences of some undesired condition q are prohibited.

In the requirements formulated for the generic PWR model, the following properties were specified using this pattern:

- The condition which triggers emergency pump activation cannot last forever.
- The following condition shall not last forever: a pressure in either of the steam generators exceeds the mean pressure by a certain value, and the emergency feed water valve closing signal is not generated.

- The following condition shall not last forever: a water level in either of the steam generators exceeds the specified value, the voltage is supplied, the emergency feedwater line is not open, and its valve opening signal is not generated.

- The following condition shall not last forever: a certain condition on pressurizer water level and mean temperature of primary circuit cold and hot legs is satisfied, and the let-down valve opening signal is not generated.

4.4 Pattern "Cannot happen until"

LTL formula	$((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$, $(\neg q)\,\mathbf{W}\,p$, $\mathbf{G}(q \rightarrow \mathbf{O}\,p)$
CTL formula	$\neg((\neg p)\,\mathbf{EU}(\neg p \wedge q))$
LTL type	safety
Semantics	q cannot happen until p (they are allowed to coincide).
Selected for formalism evaluation?	Yes

The pattern "cannot happen before" states that q cannot happen if its enabling condition p has not happened before that or on the same step. This is most naturally expressed by the temporal property $\mathbf{G}(q \rightarrow \mathbf{O}\,p)$, which uses the past-time LTL operator $\mathbf{O}$, meaning "once in the past". Using another additional operator $\mathbf{W}$ ("weak until"), which is identical to $\mathbf{U}$ except that the second argument is not required to eventually happen, the pattern can be expressed as $(\neg q)\,\mathbf{W}\,p$: "until p happens, or forever if p never happens, q must be false". By writing out the meaning of $\mathbf{W}$ explicitly, the pattern can be expressed using usual LTL operators: $((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$.

In the requirements formulated for the generic PWR model, the following properties were specified using this pattern:

- Emergency pumps activation shall not be triggered until certain dangerous values of key system parameters are reached.

- The emergency feed water line valve opening signal shall not be generated until water level in either of the steam generators exceeds the specified value.

- A valve opening signal shall not be generated until the corresponding valve is not open.

- Feedwater and steam lines shall not be closed until live steam pressure reaches a certain value.

- Emergency feedwater line opening signal shall not be generated until water level in either of the steam generators exceeds the specified value.

- Emergency feedwater line opening signal shall not be generated until voltage is supplied.

- Pressurizer spray valve open signals shall not be generated until the pressurizer pressure is sufficiently high.

4.5 Pattern "Response on the next step"

LTL formula	$\mathbf{G}(p \rightarrow \mathbf{X}\, q)$
CTL formula	$\mathbf{AG}(p \rightarrow \mathbf{AX}\, q)$
LTL type	safety
Semantics	p is always followed by q on the next step.
Selected for formalism evaluation?	Yes

Formally, the pattern "response on the next step" requires that a certain condition p causes another condition q on the next time unit. The meaning of a "time unit" can be different, and in particular its duration might even not be constant. Nevertheless, the pattern can be used to specify that the response is achieved almost immediately.

Besides, this patterns is often relevant in the context of feedback loops. If the employed modeling formalism is discrete and synchronous (e.g. NuSMV), such loops need to be explicitly resolved, which often can be done with unit delay blocks. Thus, the need of referring to values from consequent time steps emerges.

In the requirements formulated for the generic PWR model, the following property were specified using this pattern:

- If certain dangerous values of key system parameters are reached, then the reactor trip signal shall be generated on the next step.

4.6 Pattern "Infinitely often after condition"

LTL formula	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\, q)$, $(\mathbf{F}\, p) \rightarrow (\mathbf{G}\,\mathbf{F}\, q)$
CTL formula	$\mathbf{AG}(p \rightarrow \mathbf{AG}\,\mathbf{AF}\, q)$
LTL type	liveness
Semantics	Once p becomes true, q must be satisfied infinitely often in the future.
Selected for formalism evaluation?	Yes

The meaning of the pattern is almost identical to the one of "infinitely often", except that the desired condition is not obliged to happen if its activating condition p is never true.

In the requirements formulated for the generic PWR model, the following property were specified using this pattern:

- If live steam pressure once exceeds the threshold t_1, then it shall not be less than $t_2 < t_1$ forever without feedwater and steam line closing signals being generated.

4.7 Pattern "always {p[*n]} |-> q!"

LTL formula	$n = 1$: $\mathbf{G}(p \rightarrow q)$; $n = 2$: $\mathbf{G}(p \rightarrow \mathbf{X}(p \rightarrow q))$; $n = 3$: $\mathbf{G}(p \rightarrow \mathbf{X}(p \rightarrow \mathbf{X}(p \rightarrow q)))$; ...
CTL formula	$n = 1$: $\mathbf{AG}(p \rightarrow q)$; $n = 2$: $\mathbf{AG}(p \rightarrow \mathbf{AX}(p \rightarrow q))$; $n = 3$: $\mathbf{AG}(p \rightarrow \mathbf{AX}(p \rightarrow \mathbf{AX}(p \rightarrow q)))$; ...
LTL type	safety
Semantics	If p is true for n consequent steps, then q is true on the n-th of these steps.
Selected for formalism evaluation?	Yes

This pattern (or, more precisely, a series of patterns parameterized by the natural number n), which was originally specified in PSL, states that any repeated consequent satisfaction of p with the length of n leads to q on the n-th step. In LTL and CTL, this can be expressed by series of formulas with nesting increasing with the growth of n.

This pattern has never been used in the requirements formulated for the generic PWR model.

4.8 Pattern "$(\mathbf{G}\, p) \rightarrow (\mathbf{G}\, q)$"

LTL formula	$(\mathbf{G}\, p) \rightarrow (\mathbf{G}\, q)$
CTL formula	no representation
LTL type	liveness
Semantics	If p is true forever, then q is true forever.
Selected for formalism evaluation?	No

This temporal pattern states that q must be always satisfied provided that p is always true. That is, all the runs of the system are separated in two classes: the ones on which p is always satisfied (and, consequently, q is asserted to be always satisfied), and the remaining ones (with no further restrictions). This is a liveness property, since any appearance of $\neg p$ removes any restrictions from the current run. This temporal pattern can be potentially applied to specify system properties in some special cases of system execution.

This pattern has never been used in the requirements formulated for the generic PWR model.

4.9 Pattern "always {p;q[*n]} |-> r!"

LTL formula	$n = 1$: $\mathbf{G}(p \rightarrow \mathbf{X}(q \rightarrow r))$; $n = 2$: $\mathbf{G}(p \rightarrow \mathbf{X}(q \rightarrow \mathbf{X}(q \rightarrow r)))$; ...
CTL formula	$n = 1$: $\mathbf{AG}(p \rightarrow \mathbf{AX}(q \rightarrow r))$; $n = 2$: $\mathbf{AG}(p \rightarrow \mathbf{AX}(q \rightarrow \mathbf{AX}(q \rightarrow r)))$; ...
LTL type	safety
Semantics	If p is true for one step and q is true for the next n consequent steps, then r is true on the n-th of these steps.
Selected for formalism evaluation?	No

The meaning of this pattern pattern, which was originally specified in PSL, is similar to the one from Section 4.7, except that the triggering condition for the consequence r is now property p followed by n occurrences of q. Since p can be set to q, this pattern more general than the one from Section 4.7

This pattern has never been used in the requirements formulated for the generic PWR model.

4.10 Pattern "never {p[*n]}"

LTL formula	$n = 1$: $\mathbf{G}\,\neg p$; $n = 2$: $\mathbf{G}(p \rightarrow \mathbf{X}\,\neg p)$; $n = 3$: $\mathbf{G}(p \rightarrow \mathbf{X}(p \rightarrow \mathbf{X}\,\neg p)))$; ...
CTL formula	$n = 1$: $\mathbf{AG}\,\neg p$; $n = 2$: $\mathbf{AG}(p \rightarrow \mathbf{AX}\,\neg p)$; $n = 3$: $\mathbf{AG}(p \rightarrow \mathbf{AX}(p \rightarrow \mathbf{AX}\,\neg p)))$; ...
LTL type	safety
Semantics	p cannot be true for n consequent steps.
Selected for formalism evaluation?	No

This pattern (or, more precisely, a series of patterns parameterized by the natural number n), which was originally specified in PSL, states that repeated occurrences of p with the length of at least n are forbidden. In LTL and CTL, this can be expressed by series of formulas with nesting increasing with the growth of n.

This pattern has never been used in the requirements formulated for the generic PWR model.

4.11 Pattern "$\mathbf{G}(p \wedge \mathbf{X}\,q \rightarrow \mathbf{X}\,\mathbf{X}\,r)$"

LTL formula	$\mathbf{G}(p \wedge \mathbf{X}\,q \rightarrow \mathbf{X}\,\mathbf{X}\,r)$, $\mathbf{G}(p \rightarrow \mathbf{X}(q \rightarrow \mathbf{X}\,r))$
CTL formula	$\mathbf{AG}(p \rightarrow \mathbf{AX}(q \rightarrow \mathbf{AX}\,r))$
LTL type	safety
Semantics	If p is true and q is true the step after that, r is true on the following (i.e. third) step.
Selected for formalism evaluation?	No

The pattern states that any occurrence of p and q on the next step requires r on the third step. This can be interpreted similarly to the "response on the next step" pattern with the premise expressing a state change between two steps (e.g. a rising/falling edge of a certain signal).

This pattern has never been used in the requirements formulated for the generic PWR model.

4.12 Pattern "Reachable"

LTL formula	no representation
CTL formula	$\mathbf{EF}\, p$
LTL type	–
Semantics	For all initial states of the system, there is a path to a state where p is true.
Selected for formalism evaluation?	No

The pattern "reachable" specifies that from each possible initial state of the system p is potentially reachable in the future. This property has no representation in LTL, since it cannot be expressed as a general constraint on system runs.

The use of this pattern in the requirements formulated for the generic PWR model was limited to debugging the formal model of the system.

4.13 Pattern "Eventually"

LTL formula	$\mathbf{F}\, p$
CTL formula	$\mathbf{AF}\, p$
LTL type	liveness
Semantics	p is true at some time step.
Selected for formalism evaluation?	No

The pattern "eventually" specifies that in every possible run of the system p eventually happens.

The use of this pattern in the requirements formulated for the generic PWR model was limited to debugging the formal model of the system.

5 Evaluation of user-friendly formalisms on patterns

In this section, the extent is examined to which visual formalisms selected in Section 3.2 are suitable to represent temporal property patterns identified in Section 4. In particular:

- The notation of each formalism will be reviewed.
- For each visual formalism and each pattern, the pattern will be attempted to be represented using the formalism. In successful cases, the obtained representations will be shortly explained. Otherwise, the reasons of failure will be provided.

Note that many of the considered visual languages will not refer to atomic propositions (i.e. state variables of the system under verification), but rather to *events*. In this case, the following mapping of atomic propositions to events will be utilized: an event is a combination of concrete values of atomic propositions, such events happen at integer timestamps (i.e. 0, 1, 2, ...), and one after another they convey the consequent values of atomic propositions, thus providing the concrete behavior of the system. The assertion of integer timestamps is important for formalisms wherein events are able to happen at real-valued moments of time.

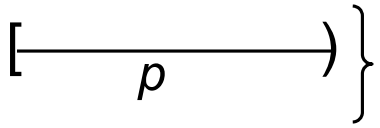
5.1 Graphical interval logic

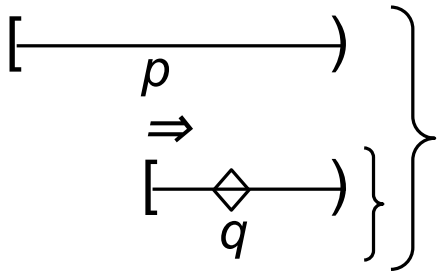
Graphical interval logic (GIL) [11] is a temporal logic whose formulas resemble timing diagrams. It is as expressive as LTL without the **X** operator. In each GIL diagram, the respective temporal property is represented on a number of horizontal left-closed right-open intervals. Such an interval has a start point but no end point, which is explained by the infinite-time semantics of LTL. Using a horizontal dashed arrow ending with a Boolean formula f, it is possible to locate the first time step where this formula is satisfied within the considered interval. The results of such searches can be used to specify new intervals. It is possible to place the following temporal restrictions on each interval:

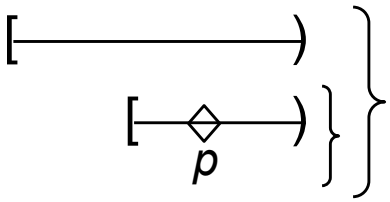
- By drawing a Boolean formula f below the left delimiter of an interval, f is asserted to hold in the start point of the interval.
- By drawing f below the center of the interval, it is asserted to hold during the whole interval. This corresponds to the **G** LTL operator.
- By drawing a diamond on the interval and drawing f below the diamond, f is asserted to hold at some point of the interval. This corresponds to the **F** LTL operator.

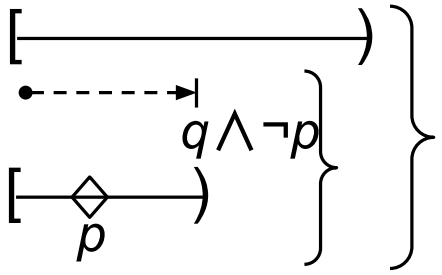
If at least one of the searches defining the interval fails, then all the restrictions are assumed to be satisfied. GIL diagrams are also permitted to have usual Boolean connectives ($\wedge, \vee, \neg, \rightarrow$). Formulas using them are composed using a vertical layout.

The temporal patterns selected in Section 4 are expressed using this formalism below:

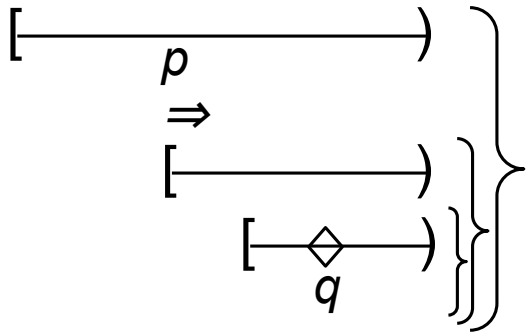
"Always"	$\mathbf{G}\,p$
	
The single interval corresponds to the entire system run, during which p is universally asserted by drawing it below the center of the interval.	

"Unbounded response"	$\mathbf{G}(p \rightarrow \mathbf{F}\,q)$
	
For each point of the entire system run, the infinite time interval starting from this point is considered. If p is true at this point, then q is asserted somewhere in the future by drawing it below the diamond.	

"Infinitely often"	$\mathbf{G}\,\mathbf{F}\,p$
	
Similar to "unbounded response", but the condition for occurrence in the future is trivial (always true).	

"Cannot happen until"	$((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$
	
A search is performed to identify the earliest time step where $q \wedge \neg p$ is true. If such a point exists, then p must have happened before. Otherwise, if the search fails, no restriction is asserted.	

“Response on the next step”	$\mathbf{G}(p \to \mathbf{X}\, q)$
No representation since the **X** operator is not supported.	

“Infinitely often after condition”	$\mathbf{G}(p \to \mathbf{G}\,\mathbf{F}\, q)$
	
Can be represented as a combination of “unbounded response” and “infinitely often” patterns.	

–	always {p[*n]} |-> q!
No representation since the **X** operator is not supported.	

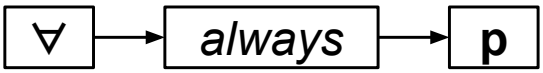
5.2 L_R temporal logic

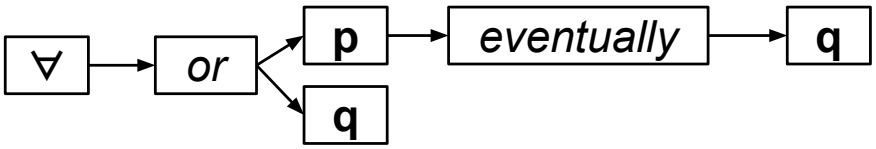
L_R [18] is a graphical temporal logic for real-time systems with discrete time. An L_R formula is a directed acyclic graph represented by a number of boxes connected with arrows. The following diagram elements are possible:

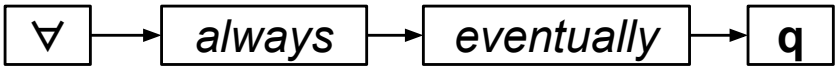
- *Predicates* are “non-recursive properties that can be decided locally for each state” [18].
- *Logical connectives*: *and*, *or*, *not*.
- *Modal operators* correspond to events in the system. L_R was designed for event-based systems, but events can be naturally replaced with atomic propositions. A modal operator refers to an occurrence of a specified event. If it is followed by some L_R subformula f, then its semantics is “whenever this event happens, for the next state of the system f must be satisfied”.
- *Path quantifiers* $\forall$ and $\exists$ specify whether the following property must be satisfied for all system executions, or for some execution. These quantifiers naturally correspond to CTL operator prefixes **A** and **E**, respectively.
- *Temporal operators* such as *eventually* and *always* (**F** and **G** in LTL, respectively), which affect the following part of the diagram. Each operator can be annotated by *path modifiers*, which may place additional restrictions on the temporal property. For example, an undesired event can be specified for the *eventually* operator using the *avoiding* modifier. Graphically, path modifiers are placed on top of arrows which link temporal operators with subformulas to which they are applied.

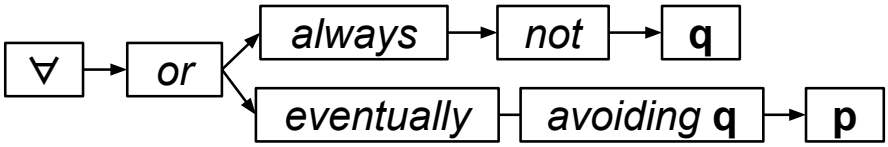
- Since in [18] the logic was suggested to be extensible via user-expert interaction, more elements defined by an expert are possible. We, however, will only use the elements mentioned above.

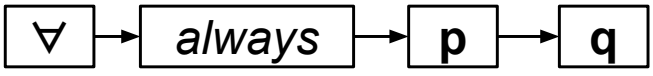
The temporal patterns selected in Section 4 are expressed using this formalism below:

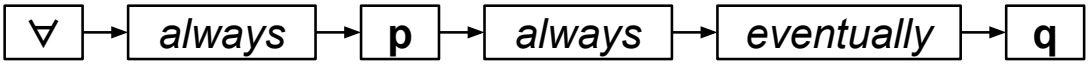
"Always"	$\mathbf{G}\,p$
	
For each path, p is always true.	

"Unbounded response"	$\mathbf{G}(p \rightarrow \mathbf{F}\,q)$
	
For each path, the *or*-formula is true. The upper branch of the *or*-formula corresponds to the LTL formula $p \rightarrow \mathbf{X}\,\mathbf{F}\,q$, and the lower part represents the missing case on encountering q already at the current time step.	

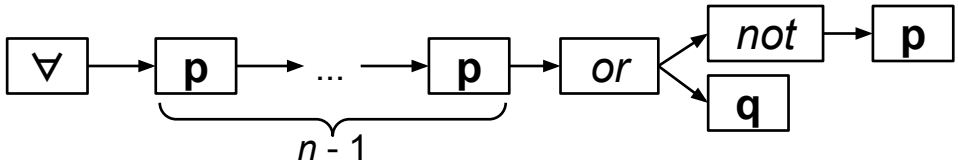
"Infinitely often"	$\mathbf{G}\,\mathbf{F}\,p$
	
For each path, q will always eventually happen in the future.	

"Cannot happen until"	$((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$
	
For each path, either q never happens, or p happens as some point without encountering q prior to that.	

"Response on the next step"	$\mathbf{G}(p \rightarrow \mathbf{X}\,q)$
	
For each path, if p happens, then it is followed by q.	

"Infinitely often after condition"	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\,q)$
	
For each path, if p happens, then it is infinitely often followed by q.	

–	always {p[*n]} |-> q!



If p happens $n-1$ times, then on the next step $\neg p \vee q$ is true.

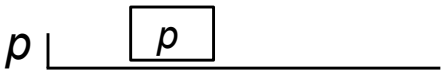
5.3 Constraint diagrams

Constraint diagrams [10] is an inspired by timing diagrams visual specification formalism for real-time requirements. Within this formalism, a requirement is represented as a number of *assumptions* and *commitments* on one or more timelines (called *waveforms*). Each waveform corresponds to a system component, or, in our case, just to a variable. Waveforms are separated into several parts by *events*, which indicate state changes. Dashed parts of waveforms refer to unspecified (i.e. unconstrained) behavior. Different waveform segments separated by events can be annotated with Boolean constraints. If a constraint is drawn without a surrounding box, it refers to the assumption part, otherwise to the commitment part (the condition which is asserted if all the assumptions are satisfied).

Waveforms can be connected with arrows expressing timing constraints over events in different waveforms. These constraints can also be attributed either to the assumption or to the commitment parts of the property. Similar constraints can be drawn below a segment of a waveform, and thus will specify timing restrictions between consecutive events within a single waveform. In general, time is allowed to be continuous, but for the purpose of expressing LTL properties it is assumed to be discrete, i.e. events only happen at integer timestamps. Constraint diagrams are formalized in the Duration Calculus temporal logic.

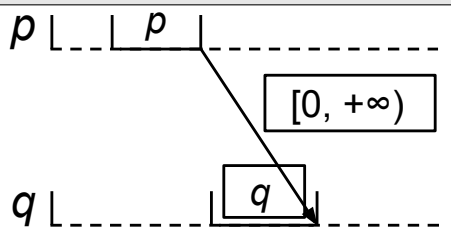
The temporal patterns selected in Section 4 are expressed using this formalism below:

"Always"	$\mathbf{G}\,p$



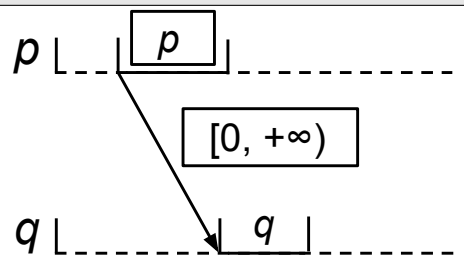
Given empty (i.e. always satisfied) assumptions, p must be satisfied forever.

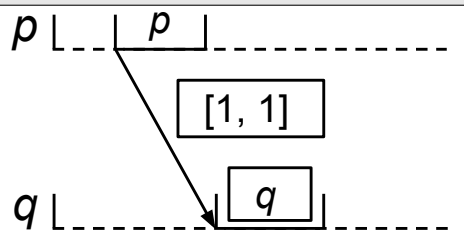
"Unbounded response"	$\mathbf{G}(p \rightarrow \mathbf{F}\,q)$



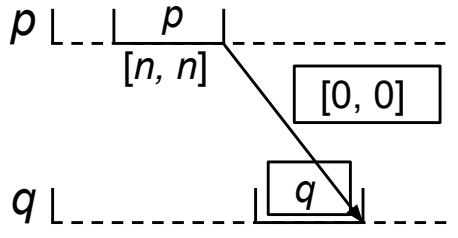
Assuming that p happens at some point, q must happen. The way p and q truth-intervals are connected allows q to happen already on the same step as p.

"Infinitely often"	$\mathbf{G}\,\mathbf{F}\,p$
	
Given empty (i.e. always satisfied) assumptions, p must be satisfied at some point in the future. Since the empty assumption is satisfied on every time step, this occurrence in the future is also asserted for every time step.	

"Cannot happen until"	$((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$
	
If q happens at some point, then p must have happened before. The way p and q truth-intervals are connected allows p to happen on the same step as q.	

"Response on the next step"	$\mathbf{G}(p \rightarrow \mathbf{X}\,q)$
	
If p happens, then q must happen strictly one time unit after that. The value of q on the time step of the occurrence of p is not important, which is indicated by a dashed line.	

"Infinitely often after condition"	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\,q)$
Supposedly, this pattern is too complex to be represented using this formalism.	

–	always {p[*n]} |-> q!
	
Every true-duration of p with length n requires q to happen on the last of these n steps (this is indicated by simultaneous ends of p- and q-intervals).	

5.4 Timing diagram editor

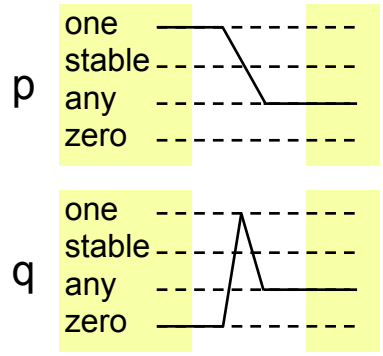
Timing diagram editor (TDE) is a tool prototype presented in [30]. It was originally supposed to be used with formal models specified as net condition/event systems (NCES), but the corresponding visual specification language can be applied to specifying temporal properties in general. Each timing diagram consists of several horizontal lanes, one for each variable involved in the temporal property. TDE can also use *events* in addition to variables, but these entities are NCES-specific and hence they are not required here. Each variable can have one of four possible levels: *zero* (false), *one* (true), *stable* (either zero or one, not changing) and *any* (either zero or one, can change arbitrarily).

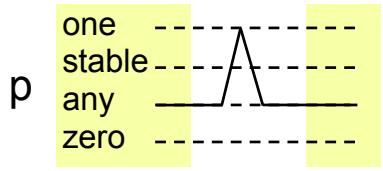
Within each lane, the *precondition* (indicated with a yellow box on the left) specifies the activation condition of a diagram. This condition may not only specify a level of the variable, but can also be represented as a change of this level. The main part of the diagram is drawn on the right from the precondition. It may indicate any variable level changes. The changes of these levels in different lanes can be bounded by order restrictions ($=$, $\neq$, $>$) drawn in circles. Otherwise, the order of events on different lanes is unspecified, and their relative horizontal offset is not taken into account. Finally, the *postcondition* (indicated by a yellow box on the right) is similar to the precondition, but specifies the condition which ends the checking process.

Each diagram has six possible semantics. Thus, an attempt to interpret a diagram without knowing its semantics may lead to confusion. To specify a particular property, one has to both create a timing diagram and indicate its intended semantics. The semantics which will be used to represent patterns requires that every occurrence of a precondition constrains the behavior of the model according to the main part of the diagram, until the postcondition is reached.

The temporal patterns selected in Section 4 are expressed using this formalism below:

"Always"	$\mathbf{G}\,p$
No representation, since any precondition has length of at least 1, and thus it is impossible to require p for the first step of system execution. A possible workaround might comprise a diagram which specifies an unsatisfiable property triggered by $\neg p$. This solution is not provided here due to its unnaturality.	

"Unbounded response"	$\mathbf{G}(p \rightarrow \mathbf{F}\,q)$
	
At every time step, if p is true and q is false, q must become true in the future and p is allowed to behave arbitrarily.	

“Infinitely often”	$\mathbf{G}\,\mathbf{F}\,p$
	
At every time step, p must become true in the future. $\mathbf{F}\,p$ is specified after the trivial precondition, so the literal meaning of this diagram is $\mathbf{G}\,\mathbf{X}\,\mathbf{F}\,p$, but this formula is equivalent to $\mathbf{G}\,\mathbf{F}\,p$.	

“Cannot happen until”	$((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$
No representation, since this requirement at least forbids $q \wedge \neg p$ on the first step, which is impossible to specify (see the comment for the “always” pattern).	

“Response on the next step”	$\mathbf{G}(p \rightarrow \mathbf{X}\,q)$
No representation since the $\mathbf{X}$ operator is not supported.	

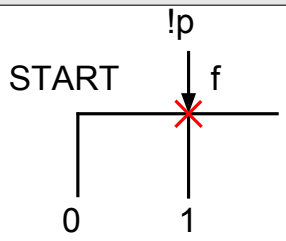
“Infinitely often after condition”	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\,q)$
Supposedly, this pattern is too complex to be represented using this formalism.	

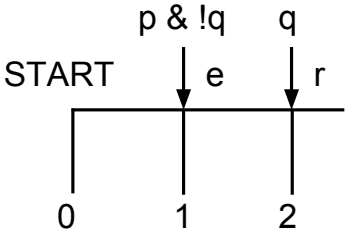
–	always {p[*n]} |-> q!
No representation since the $\mathbf{X}$ operator is not supported.	

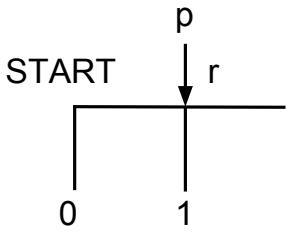
5.5 TimeLine editor

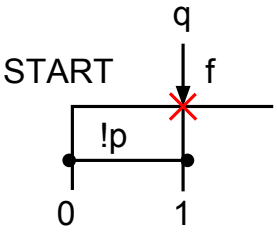
TimeLine editor [28] is a tool which is able to specify temporal requirements comprising a subset of the LTL logic. In particular, a transformation of timeline diagrams into Büchi automata, which can also be used to represent LTL formulas, is presented in [28]. A timeline diagram is a horizontal lane, where discrete time flows from left to right. The left of the lane corresponds to the initial time step. Then, a number of *events* can occur along the timeline. Events are subdivided in three types: *regular* events specify activation conditions of property checking, *required* events are asserted to happen if all regular events on the left have happened, and *fail* events are forbidden to happen given the same precondition. Each event is indicated by a vertical line, an arrow on top of this line, the name of the event (in our case, a Boolean formula specifying it), and its type (“e”, “r” or “f” respectively). For the purpose of this report, it is possible to interpret events as Boolean variables. In addition, *constraints*, drawn as black horizontal lines and annotated with asserted Boolean formulas, exclude scenarios violating them from consideration. A constraint may either include a neighboring event (indicated as a filled circle) or not (indicated as a vertical bar).

The temporal patterns selected in Section 4 are expressed using this formalism below:

"Always"	$\mathbf{G}\,p$
	
$\mathbf{G}\,p$ is asserted by prohibiting $\neg p$ (using a fail event). There are no regular events, so this prohibition is valid universally.	

"Unbounded response"	$\mathbf{G}(p \rightarrow \mathbf{F}\,q)$
	
If $p \wedge \neg q$ is true (regular event), then q must happen in the future (required event).	

"Infinitely often"	$\mathbf{G}\,\mathbf{F}\,p$
	
Given an empty precondition (which is true at every time step), p must happen in the future (required event).	

"Cannot happen until"	$((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$
	
The following situation is prohibited: q happens (fail event), and until this moment, including the moment when q happens, p has not been encountered (this constraint is indicated with a horizontal line with circles on both ends).	

"Response on the next step"	$\mathbf{G}(p \rightarrow \mathbf{X}\,q)$
No representation since the $\mathbf{X}$ operator is not supported.	

"Infinitely often after condition"	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\,q)$
Supposedly, this pattern is too complex to be represented using this formalism.	

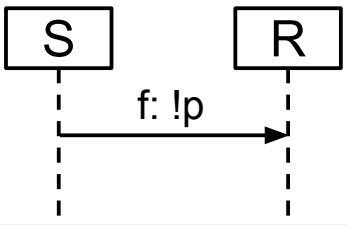
–	always {p[*n]} |-> q!
No representation since the **X** operator is not supported.	

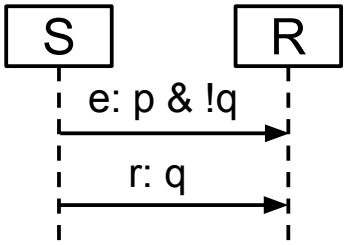
5.6 Property sequence charts

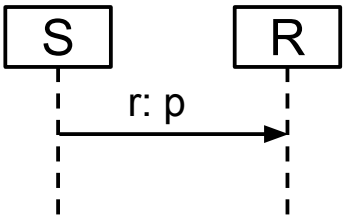
Property sequence charts (PSCs) [4] is a graphical formalism for expressing a subset of LTL properties. It has been partially inspired by the TimeLine Editor [28], but supports more general ways of specifying constraints and additional features such as alternative or repeated execution. The visual basis for PSCs is UML sequence diagrams, which means than events now correspond to message exchanges between the components of the system. As in the case of the TimeLine Editor, events are replaced with Boolean formulas over the state space of the system. Since specific system components are not important in the present report, message exchanges between two dummy components, the sender (S) and the receiver (R), will be considered.

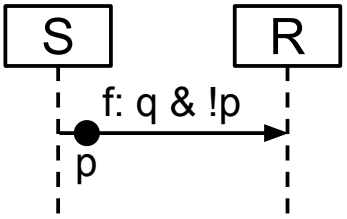
A PSC consists of a number of vertical *lifelines* for each considered component. On top of lifelines are component names written in boxes. Lifelines extend downwards according to the flow of time. On the lifelines, horizontal arrows indicate message submissions between components (in our case, the only possible message type is from the sender to the receiver). Similarly to the TimeLine Editor, messages can be either *regular*, *required* or *fail* ones (message type semantics and letter abbreviations are kept). *Strict ordering* of messages can be specified by connecting two submission points of consecutively drawn messages with a bold vertical line segment. This prohibits any message submissions between the drawn messages, thus allowing us to express the **X** LTL operator, which was generally impossible in the TimeLine Editor. In addition, each message submission can be annotated with *constraints*. Many types of constraints are supported, but among them only the *past unwanted message constraint* will be needed, which is drawn as a filled circle on the left of the message arrow. The exact semantics of this constraint varies depending on the message type, but its main purpose is to disallow certain events before the previous one on the diagram and the current one.

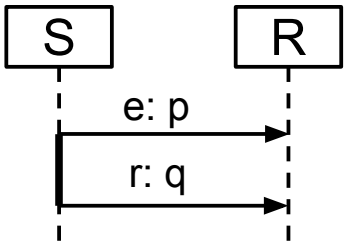
The temporal patterns selected in Section 4 are expressed using this formalism below:

"Always"	$\mathbf{G}\,p$
	
$\mathbf{G}\,p$ is asserted by prohibiting $\neg p$ (using a fail event). There are no regular events, so this prohibition is valid universally.	

"Unbounded response"	$\mathbf{G}(p \rightarrow \mathbf{F}\, q)$
	
If $p \wedge \neg q$ is true (regular event), then q must happen in the future (required event).	

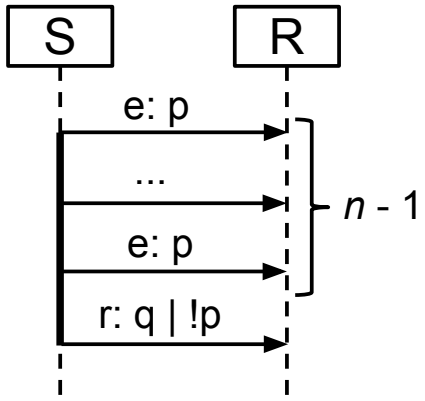
"Infinitely often"	$\mathbf{G}\,\mathbf{F}\, p$
	
Given an empty precondition (which is true at every time step), p must happen in the future (required event).	

"Cannot happen until"	$((\neg q)\,\mathbf{U}\, p) \vee \mathbf{G}\, \neg q$
	
The following situation is prohibited: $q \wedge \neg p$ happens (fail event), and until this moment p has not been encountered (this constraint is indicated with a filled circle on the left of the fail event arrow).	

"Response on the next step"	$\mathbf{G}(p \rightarrow \mathbf{X}\, q)$
	
If p happens (regular event), then on the next step q happens (required event). The contiguity of p and q is indicated with the bold line segment on the left.	

"Infinitely often after condition"	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\, q)$
Supposedly, this pattern is too complex to be represented using this formalism.	

–	always {p[*n]} |-> q!



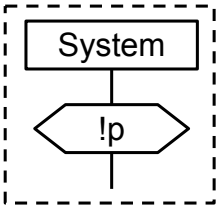
After $n-1$ contiguous occurrences of p (required events), $q \vee \neg p$ is asserted on the next step (required event).

5.7 Live sequence charts

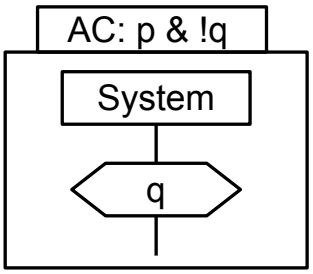
Live sequence charts (LSCs) [9] are built upon well-known message sequence charts (MSCs), standardized by the International Telecommunication Union (ITU). Since MSCs visually took similar to UML sequence diagrams, live sequence charts resemble previously reviewed PSCs (Section 5.6). Compared to MSCs, in LSCs specific effort has been put in distinguishing between optional and required events and expressing liveness properties. The implementations of these features are, however, different than the ones of PSCs. Conditional message submissions (indicated as arrows) and state conditions (indicated as hexagons) are drawn in dashed lines (and are called *cold*), while required (*hot*) ones are drawn in solid lines. Since system modularity is not important for us, only state conditions will be used to represent temporal property patterns, and only one dummy component "System" will be considered. The whole chart can be either *existential*, which means that the described execution scenario must be possible, and *universal*. Especially for universal charts, *activation conditions* are important. The semantics of universal LSCs is as follows: the described scenario must be satisfied in the future whenever the activation condition is met.

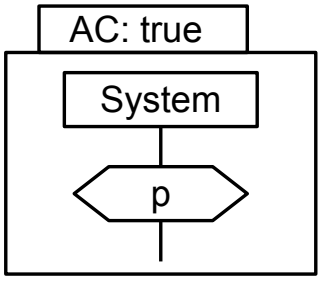
The temporal patterns selected in Section 4 are expressed using this formalism below:

"Always"	$\mathbf{G}\,p$



This LSC actually specifies the **negation** of the pattern: there exists a reachable state of the system where $\neg p$ is true. This chart is existential, which is indicated with its dashed frame.

"Unbounded response"	$\mathbf{G}(p \rightarrow \mathbf{F}\, q)$
	
Whenever the activation condition $(p \wedge \neg q)$ is met, q must happen in the future. This chart is universal, which is indicated with its solid frame.	

"Infinitely often"	$\mathbf{G}\,\mathbf{F}\, p$
	
Whenever the trivial activation condition is met (i.e. always), p must happen in the future. This chart is universal, which is indicated with its solid frame.	

"Cannot happen until"	$((\neg q)\,\mathbf{U}\, p) \vee \mathbf{G}\, \neg q$
Supposedly, this pattern is too complex to be represented using this formalism.	

"Response on the next step"	$\mathbf{G}(p \rightarrow \mathbf{X}\, q)$
No representation since the **X** operator is not supported.	

"Infinitely often after condition"	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\, q)$
Supposedly, this pattern is too complex to be represented using this formalism.	

–	always {p[*n]} |-> q!
No representation since the **X** operator is not supported.	

5.8 Symbolic timing diagrams

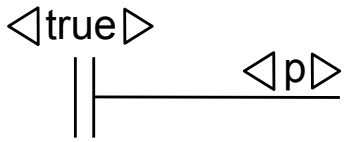
Symbolic timing diagrams (STDs) [25] graphically represent a subset of LTL. Only a subset of this logic called linear STDs will be used here; nevertheless, general STDs can always be represented using only linear STDs. Each property is specified using a horizontal line which refers to the time. The beginning of the line, which is the entry condition of the property to be specified, can be marked either with a single or double vertical line annotated with a temporal formula (if the formula is written explicitly, then it is placed inside triangular brackets). A single vertical line refers to any moment of time, while a double vertical line refers to the initial step of the

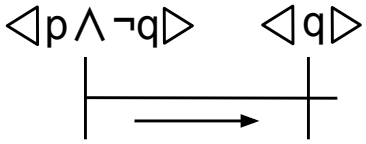
system execution path.

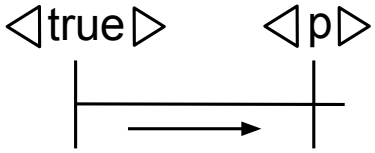
Further along the line, more vertical lines are possible, so the horizontal line is separated in a number of segments. Conditions written above segments are Boolean formulas which are asserted during these segments. Following that, non-initial vertical lines, which are also marked with Boolean conditions, indicate moments when the corresponding conditions become satisfied. These moments can be mandatory to eventually happen (thus, liveness properties can be specified): this is indicated by a horizontal arrow below the preceding horizontal segment.

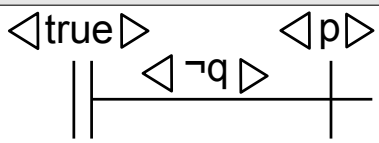
The evaluation of the diagram against a specific system execution path proceeds as follows. Diagram checking is started either in the beginning of the path (if the initial condition is false, the specification is violated) or, in the case of a single vertical line, at every step when the initial condition is satisfied. Until a new vertical line is reached (or forever, if there are no more vertical lines), the segment condition is checked. If both the segment condition and vertical line condition to the right (if present) are false, then the specification is violated. It is also violated if the vertical line on the right is never reached and it has been indicated as mandatory by an arrow below the current segment.

The temporal patterns selected in Section 4 are expressed using this formalism below:

"Always"	$\mathbf{G}\,p$
	
Starting from the beginning of system execution (indicated by the double line with the trivial condition), p must be universally satisfied.	

"Unbounded response"	$\mathbf{G}(p \rightarrow \mathbf{F}\,q)$
	
Whenever $p \wedge \neg q$ happens, q must eventually happen (this is indicated with the horizontal arrow).	

"Infinitely often"	$\mathbf{G}\,\mathbf{F}\,p$
	
Whenever the trivial condition is satisfied (i.e. always), p must eventually happen (this is indicated with the horizontal arrow).	

“Cannot happen until”	$((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$
	
Starting from the beginning of system execution (indicated by the double line with the trivial condition), $\neg q$ must be satisfied until p happens. The occurrence of p is not required, and in this case the assertion $\neg q$ must be satisfied forever.	

“Response on the next step”	$\mathbf{G}(p \rightarrow \mathbf{X}\,q)$
No representation since the **X** operator is not supported.	

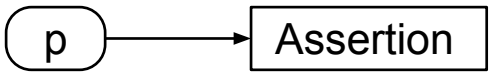
“Infinitely often after condition”	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\,q)$
Supposedly, this pattern is too complex to be represented using this formalism.	

–	always {p[*n]} |-> q!
No representation since the **X** operator is not supported.	

5.9 Simulink design verifier

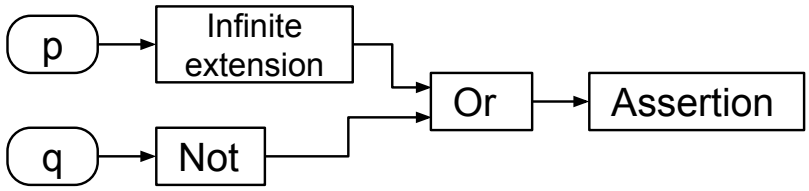
Simulink design verifier [20] is a tool capable of modeling and analyzing functional and safety requirements for discrete time systems. This approach is appealing for the I&C domain since requirements are represented using function blocks, which are common in this domain. Supported function blocks include simple transformations, such as Boolean operators, and special function blocks with memory serving as substitutes for temporal operators. In particular, the *extender* will be needed, which extends the truth value of a Boolean signal for a certain number of steps or infinitely. Liveness properties are not supported.

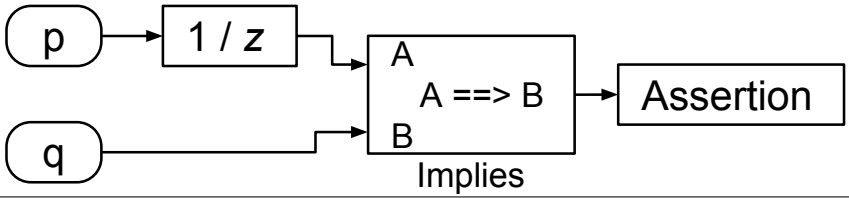
The temporal patterns selected in Section 4 are expressed using this formalism below:

“Always”	$\mathbf{G}\,p$
	
p is always asserted.	

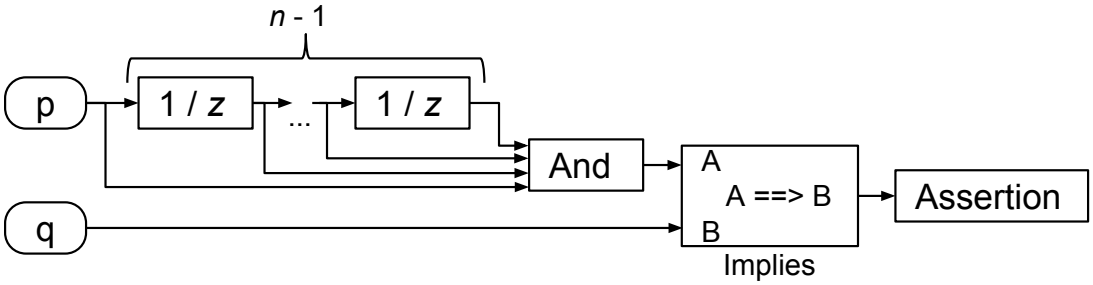
“Unbounded response”	$\mathbf{G}(p \rightarrow \mathbf{F}\,q)$
No representation since liveness properties are not supported.	

“Infinitely often”	$\mathbf{G}\,\mathbf{F}\,p$
No representation since liveness properties are not supported.	

"Cannot happen until"	$((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$
	
The following condition is always asserted: either q has not happened yet, or p has happened in the past (the infinite extender block detects this).	

"Response on the next step"	$\mathbf{G}(p \rightarrow \mathbf{X}\,q)$
	
It is asserted that p, delayed in 1 time unit, implies q.	

"Infinitely often after condition"	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\,q)$
No representation since liveness properties are not supported.	

–	always {p[*n]} |-> q!
	
It is asserted that the conjunction of p and its $n-1$ previous values (obtained using unit delays) implies q.	

5.10 Visual timed event scenarios

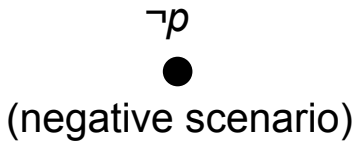
Visual timed event scenarios (VTS) [1] is a graphical formalism which is able to express constraints over the order of events. VTS were intended to be used to specify properties for timed automata. To apply this formalism to Kripke structures, each new event will be assumed to happen at the next integer value of the global clock. As usual, when temporal patterns will be expressed, events will be replaced with Boolean formulas.

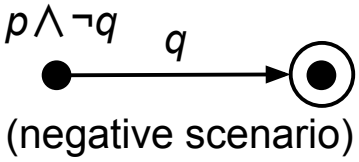
An event is represented as a filled circle. Special circles correspond to the beginning of system execution (a large circle) and to its end (a bull's-eye symbol). An arrow from one event to another asserts that the second event happens after the first one. Such an arrow can be annotated with another event, which is assumed to be prohibited between the first and the second event. An arrow can be also marked

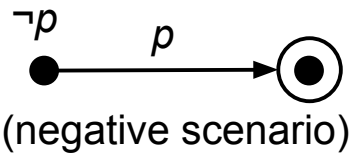
with more arrowheads within its body. For example, two arrowheads facing opposite directions indicate that the first and the second event are strictly consecutive ones.

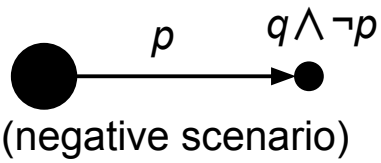
Thus, a VTS diagram consists of circles, arrows and various annotations. This graphical language supports even more entities, but they are omitted here since they are not needed to represent the temporal patterns. Each diagram can be interpreted either as a positive scenario (i.e. each system execution must match this scenario) or as a negative scenario (i.e. each system execution must not match this scenario). The latter option will be used ubiquitously.

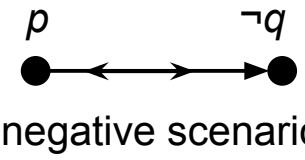
The temporal patterns selected in Section 4 are expressed using this formalism below:

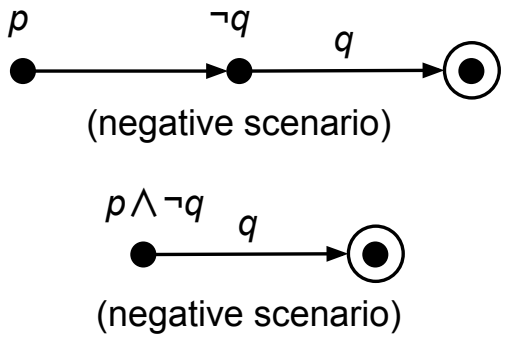
"Always"	$\mathbf{G}\,p$
	
The scenario where $\neg p$ occurs is undesired.	

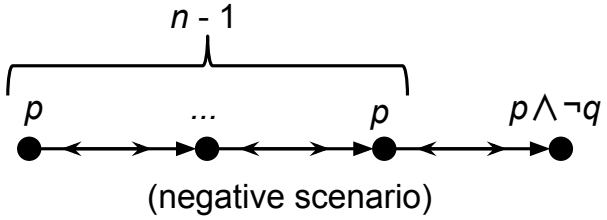
"Unbounded response"	$\mathbf{G}(p \rightarrow \mathbf{F}\,q)$
	
The following scenario is undesired: $p \wedge \neg q$ occurs, and q does not happen until the end of the execution.	

"Infinitely often"	$\mathbf{G}\,\mathbf{F}\,p$
	
The following scenario is undesired: $\neg p$ occurs, and p does not happen until the end of the execution.	

"Cannot happen until"	$((\neg q)\,\mathbf{U}\,p) \vee \mathbf{G}\,\neg q$
	
The following scenario is undesired: $q \wedge \neg p$ occurs, and there has not been any p starting from the beginning of the execution.	

"Response on the next step"	$\mathbf{G}(p \to \mathbf{X}\, q)$
	
The following scenario is undesired: p and then $\neg q$ occur strictly one after another.	

"Infinitely often after condition"	$\mathbf{G}(p \to \mathbf{G}\,\mathbf{F}\, q)$
	
The pattern is expressed using two negative scenarios. The first one prohibits the following situation: p happens, then some time after $\neg q$ happens, and q does not happen forever. The second scenario is similar to the first one, except that p and $\neg q$ happen simultaneously.	

–	always {p[*n]} |-> q!
	
The following scenario is undesired: during n consecutive time units, p is true, and q is false at the last time unit.	

5.11 Other approaches

In this subsection, several other visual approaches are mentioned, and the explanations why they were not chosen for detailed analysis are provided.

A *domain ontology* is a way to represent knowledge in a graphical way: as concepts (represented as vertices of a graph) and relations between them (represented as edges). An example of applying these ideas to industrial automation is the work [26]. Domain ontologies are also applicable for model checking [19]. However, developing such an ontology requires effort, and this approach cannot be applied if one needs just a means of expressing temporal properties in a user-friendly way.

Visual contract language and the tool *Visual contract builder* [3] which supports it are applicable for visual design modeling of software. The tool is based on formal methods; however, it does not support model checking. The tool is also oriented on

object-oriented design, which is not relevant in the context of this report.

Data flow diagrams have been applied in the context of formal specifications in [15]. However, data flow modeling is not among the features need for visual property specification in this report: the considered temporal patterns regulate the behavior of at most several variables. In addition, no connections of data flow diagrams with model checking have been given in [15].

The work [24] proposes an approach to visualize executable Z specifications using animation. Although Z specifications can be model-checked [27], the Z formalism was not designed to specify temporal properties.

6 Analysis

In this section, the findings of Section 5 are summarized, and the ten selected visual specification languages are analyzed with respect to the temporal patterns they can represent. Table 3 shows how many patterns from Section 4 each visual formalism supports. In addition to the total number of patterns, these numbers are provided separately for safety patterns ("always", "cannot happen until", "response on the next step", "always p[*n] |-> q!"), liveness patterns ("unbounded response", "infinitely often", "infinitely often after condition"), and indicate whether the **X** LTL operator (used in the patterns "response on the next step" and "always p[*n] |-> q!") is supported by the formalism.

Table 3: Comparison of visual specification languages.

Visual language	Safety	Liveness	Sum	Next operator support
Graphical interval logic	2/4	3/3	5/7	+
L_R temporal logic	4/4	3/3	7/7	+
Constraint diagrams	4/4	2/3	6/7	+
Timing diagram editor	0/4	2/3	2/7	−
TimeLine editor	2/4	2/3	4/7	−
Property sequence charts	4/4	2/3	6/7	+
Live sequence charts	1/4	2/3	3/7	−
Symbolic timing diagrams	2/4	2/3	4/7	−
Simulink design verifier	4/4	0/3	4/7	+
Visual timed event scenarios	4/4	3/3	7/7	+

According to Table 3, the formalisms which are able to represent all the considered temporal property patterns are L_R temporal logic and visual timed event scenarios. L_R achieves this result because of its ability to graphically represent temporal formulas in a structured way, by associating temporal operators, logical connectives and atomic propositions with boxes which are connected with arrows according to the structure of the temporal formula. The negative side of this, however, is the lack of any consideration of timelines, which are natural to represent requirements involving time and are used in the majority of other formalisms. This raises a concern that L_R might not be as intuitive as other formalisms. Next, visual timed event scenarios do not use timelines as well, but in the diagrams specified in this language consecutive events are connected with arrows, which conveys the reference to time.

Several other formalisms are also worth considering for specification representation in the I&C domain. Constraint diagrams, property sequence charts and Simulink design verifier succeeded in representing all four safety patterns. These patterns can be considered as more important than liveness ones, since liveness properties usually can be transformed to safety ones by specifying concrete time limits (for example, by specifying the time allowed for the response in a formula matching the "unbounded response" pattern), although performing such transformations might not be considered user-friendly. The benefit of the first two formalisms is the use of intuitive timelines, and the third one specifies properties using function blocks, which are natural for automation engineers.

Note that the examined visual formalisms are not required to be used for property formalization in the I&C industry as is: some of them can be potentially improved to match this domain better, or be used as a basis for a new visual language. Below, several suggestions to improve some of these formalisms are provided:

- In symbolic timing diagrams and the TimeLine editor, the support of the **X** operator can be added. The corresponding translations of these languages to Büchi automata can be generalized to support this operator.
- Many visual formalisms, such as TimeLine editor and visual timed event scenarios, are based on the concept or events. As mentioned in the beginning of Section 5, events can be replaced by tuples of variable values. Speaking of variable values without mentioning events can improve the intuitivity of representing specifications whose natural language representation does not involve events.
- Message passing, which is not required for the purposes of this report, can be omitted from property sequence charts.
- The support of empty preconditions can be added to timing diagram editor to facilitate the support of safety properties.

Finally, to see how concrete safety-related requirements can be represented with the mentioned languages, one should replace variable names p and q found in representation examples (Section 5) with real variable names or Boolean formulas expressing the essence of the requirements, examples of which for the generic PWR model can be found in Section 4. For example, consider the requirement "if steam pressure in either of the two steam generators exceeds the mean pressure by 4 bars and emergency feedwater lines are not closed, then the corresponding valve closing signals shall eventually be generated", which can be represented using two instances of the "unbounded response" pattern ($\mathbf{G}(p \rightarrow \mathbf{F}\, q)$), one for each valve closing signal. Assume that pressures in steam generators are modeled as integer variables p_1 and p_2, $p_m = (p_1 + p_2)/2$ is the mean pressure, the Boolean variable c indicates whether the feedwater lines are closed, and v_1 and v_2 are Boolean variables representing valve closing signals. Then p can be substituted with $(p_1 - p_m > 4) \wedge \neg c$ or $(p_2 - p_m > 4) \wedge \neg c$ (depending on the steam generator for which the requirement is formulated), and q can be replaced with either v_1 or v_2. In Fig. 1–3, the aforementioned requirement for the first steam generator is expressed in L_R temporal logic [18] (Section 5.2), visual timed event scenarios [1] (Section 5.10) and property sequence charts [4] (Section 5.6).

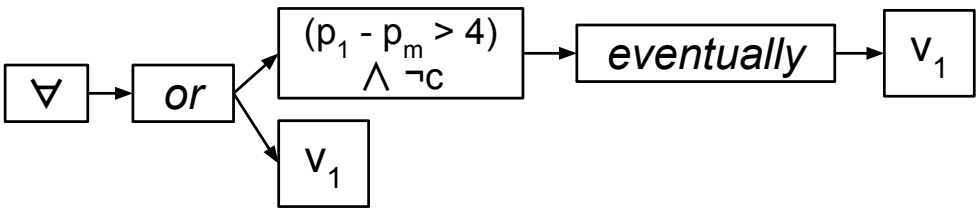


Figure 1: Example of a requirement formulated in L_R temporal logic [18].

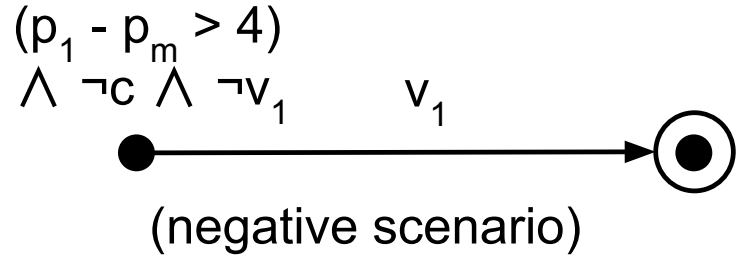


Figure 2: Example of a requirement formulated in visual timed event scenarios [1].

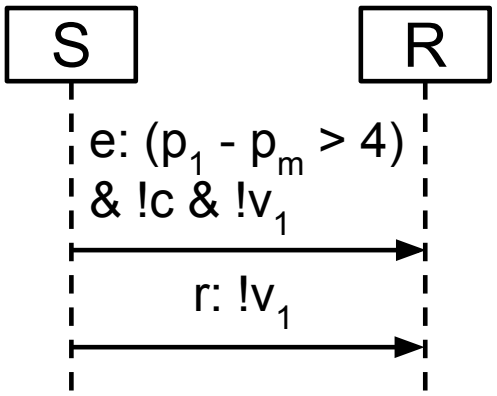


Figure 3: Example of a requirement formulated in property sequence charts [4].

7 Conclusions

This report continues one of the subgoals of the SAUNA project: to find means of representing formal specifications in a user-friendly way, and in particular in a way which can be understood by the stakeholders in the nuclear industry. Based on previous research and on specifications formulated for the generic PWR model in Apros, seven temporal specification patterns common for the nuclear I&C industry have been selected, and their use has been illustrated. Then, ten existing visual specification languages have been evaluated with respect to their ability to express these patterns. As a result, the following visual formalisms have been found possible to be potentially applied or employed as a basis for a new property specification language:

- L_R temporal logic [18] (Section 5.2);
- visual timed event scenarios [1] (Section 5.10);
- constraint diagrams [10] (Section 5.3);
- property sequence charts [4] (Section 5.6);
- Simulink design verifier [20] (Section 5.9).

Future work in the direction of the report may concern the development of either a tool or a visual language tailored for the nuclear automation domain. The selected existing property specification languages may serve as the basis for this purpose. In addition to temporal property formalization, this tool or language may incorporate a way to visually represent counterexamples to unsatisfied temporal properties. As far as the author knows, the problem of user-friendly counterexample representation has by now been ignored by researchers.

References

[1] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual timed event scenarios. In *International Conference on Software Engineering (ICSE 2004)*, pages 168–177. IEEE Computer Society, 2004.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[3] N. Amálio and C. Glodt. A tool for visual and formal modelling of software designs. *Science of Computer Programming*, 98:52–79, 2015.

[4] M. Autili, P. Inverardi, and P. Pelliccione. Graphical scenarios for specifying temporal properties: an automated approach. *Automated Software Engineering*, 14(3):293–340, 2007.

[5] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT press, 2008.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[7] I. Buzhinsky and V. Vyatkin. Plant model inference for closed-loop verification of control systems: Initial explorations. In *IEEE International Conference on Industrial Informatics (INDIN 2016)*, pages 18–21. IEEE, 2016.

[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[9] W. Damm and D. Harel. Lscs: Breathing life into message sequence charts. *Formal methods in system design*, 19(1):45–80, 2001.

[10] C. Dietz. Graphical formalization of real-time requirements. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 366–384. Springer, 1996.

[11] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(2):131–165, 1994.

[12] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In *2nd Workshop on Formal methods in Software Practice*, pages 7–15. ACM, 1998.

[13] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering (ICSE 1999)*, pages 411–420. IEEE, 1999.

[14] K. Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, 1996.

[15] R. B. France. Semantically extended dataflow diagrams: a formal specification tool. *IEEE Transactions on Software Engineering*, 18(4):329–346, 1992.

[16] G. J. Holzmann. The logic of bugs. In *10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 81–87. ACM, 2002.

[17] D. Kleyko, E. Osipov, N. Papakonstantinou, V. Vyatkin, and A. Mousavi. Fault detection in the hyperspace: Towards intelligent automation systems. In *IEEE International Conference on Industrial Informatics (INDIN 2015)*, pages 1219–1224. IEEE, 2015.

[18] I. Lee and O. Sokolsky. A graphical property specification language. In *High-Assurance Systems Engineering Workshop*, pages 42–47. IEEE, 1997.

[19] Z.-y. Li, Z.-x. Wang, A.-h. Zhang, and Y. Xu. The domain ontology and domain rules based requirements model checking. *International Journal of Software Engineering and Its Applications*, 1(1), 2007.

[20] MathWorks. Simulink design verifier: Identify and isolate design errors and generate tests. Available at: https://se.mathworks.com/products/sldesignverifier/, 2016.

[21] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela. A toolset for model checking of PLC software. In *IEEE Conference on Emerging Technologies & Factory Automation (ETFA 2013)*, pages 1–6. IEEE, 2013.

[22] A. Pakonen, C. Pang, I. Buzhinsky, and V. Vyatkin. User-friendly formal specification languages – conclusions drawn from industrial experience on model checking. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*, pages 1–8. IEEE, 2016.

[23] C. Pang, A. Pakonen, I. Buzhinsky, and V. Vyatkin. A study on user-friendly formal specification languages for requirements formalization. In *IEEE International Conference on Industrial Informatics (INDIN 2016)*, pages 676–682. IEEE, 2016.

[24] P. Parry, I. Morrey, J. Siddiqi, et al. Visualisation of executable formal specifications for user validation. In *Services and Visualization Towards User-Friendly Design*, pages 142–157. Springer, 1998.

[25] R. C. Schlör. *Symbolic timing diagrams: A visual formalism for model verification*. PhD thesis, Universität Oldenburg, 2002.

[26] R. Sinha, C. Pang, G. S. Martínez, J. Kuronen, and V. Vyatkin. Requirements-aided automatic test case generation for industrial cyber-physical systems. In *20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 198–201. IEEE, 2015.

[27] G. Smith and K. Winter. Proving temporal properties of z specifications using abstraction. In *International Conference of B and Z Users*, pages 260–279. Springer, 2003.

[28] M. H. Smith, G. J. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *5th IEEE International Symposium on Requirements Engineering*, pages 14–22. IEEE, 2001.

[29] T. Tommila and A. Pakonen. Controlled natural language requirements in the design and analysis of safety critical I&C systems. Technical report, VTT Research Centre of Finland, January 2014.

[30] V. Vyatkin and G. Bouzon. Using visual specifications in verification of industrial automation controllers. *EURASIP Journal on Embedded Systems*, 2008(1):1–9, 2007.