The Journal of Systems and Software 216 (2024) 112153

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

In Practice

Evaluation of visual property specification languages based on practical model-checking experience☆,☆☆

Antti Pakonen [a,*], Igor Buzhinsky [b], Valeriy Vyatkin [c,d]

[a] VTT Technical Research Centre of Finland Ltd., Espoo, Finland
[b] IPRally Technologies Oy, Helsinki, Finland
[c] Department of Electrical Engineering and Automation, Aalto University, Espoo, Finland
[d] Department of Computer Science, Computer and Space Engineering of Luleå Tekniska Universitet, Sweden

ARTICLE INFO

Dataset link: Formal properties collected from practical model-checking projects (Original data)

Keywords:
Formal specifications
Instrumentation and control
Formal verification
Model checking
Requirements engineering

ABSTRACT

Formal verification methods like model checking can provide mathematical proofs of design correctness, so their use is justified in applications where safety or reliability requirements are high. A key challenge for the wider adoption of model checking is the effort and expertise needed in formalizing functional requirements into verifiable properties. A particular challenge in specifying formal properties for industrial instrumentation and control (I&C) logics is accounting for the sequencing and timing issues that arise from, e.g., the dynamic behavior of the plant being controlled. In this paper, we evaluate different visual property specification languages that are aimed at making formal methods more accessible. We have collected 3923 formal properties from practical model checking projects in the nuclear and rail traffic industries and identified the most commonly occurring types of properties. Based on the sample data, a real-world example logic, and our practical experience, we identify requirements for a user-friendly property specification language most suited for our specific domain of industrial I&C.

1. Introduction

Due to the complexity of modern, digital industrial instrumentation and control (I&C) systems, conventional verification methods like testing, simulation, and manual reviews are not enough to guarantee that the designs are error-free. In safety-critical domains like the nuclear or the rail traffic industries, we need more rigorous proofs for the safety and reliability of I&C system logics. And even if the application is not safety-critical, reliability requirements can be very high, if the cost of production loss due to I&C failures is excessive.

Formal methods provide a rigorous and systematic framework for requirement specification and design activities. Such a framework helps demonstrate the consistency and completeness of systems specifications (IAEA, 2021). Requirement specification is a critical activity for project success, and formal specifications — written in a language with strict grammar rules and precise semantics — enable rigorous analysis techniques and mathematical proofs through the use of formal verification methods (Bel V et al., 2018).

Model checking (Clarke et al., 1999) is a formal verification method, based on mathematical proof that system model satisfies the stated formal specifications. In Finland, VTT[1] has used the method for over a decade to verify I&C logics in the nuclear (Pakonen et al., 2021a) and rail traffic industries.

The catch is that formalization of system requirements into automatically verifiable properties is hard and error-prone (Schlör et al., 1998). The use of formalisms such as temporal logics requires a level of sophistication many users never develop (Holzmann, 2002). The training required to reach proficiency may outweigh expected benefits (Bel V et al., 2018), which presents an impediment to moving formal techniques from theoretical research to practice (Autili et al., 2007). If used inappropriately, by insufficiently trained personnel, or without proper tool support, formal methods can even be dangerous (Bel V et al., 2018).

Since the 1990s, there have been different attempts to create a user-friendly specification language, to hide the complexity of the underlying formal language, and allow the user work with a more acceptable way of representation—be it a pattern, a natural language template, or a visual language, preferably one that resembles a

☆ This work was supported by the Finnish Research Programme on Nuclear Power Plant Safety 2018–2022 (SAFIR 2022).
☆☆ Editor: Antonio Filieri.
* Corresponding author.
E-mail addresses: antti.pakonen@vtt.fi (A. Pakonen), igor.buzhinsky@gmail.com (I. Buzhinsky), valeriy.vyatkin@aalto.fi (V. Vyatkin).
[1] VTT Technical Research Centre of Finland Ltd. is a state owned company providing research and innovation services. https://www.vttresearch.com/

https://doi.org/10.1016/j.jss.2024.112153
Received 9 August 2023; Received in revised form 9 February 2024; Accepted 27 June 2024
Available online 10 July 2024

language already used in the industry. However, no standard approach has so far emerged.

This paper is an extended version of our previous works (Pakonen et al., 2016) and Buzhinsky (2018). We extend the earlier works through a larger dataset, a larger number of covered languages, and a real-world example. The contribution is fourfold. First, we publish and analyze a set of 3923 formal properties collected from VTT's practical model-checking projects. Second, we present a running example, where we reveal two real-world design issues VTT has detected in nuclear I&C systems. Third, based on our practical experience and the review work presented here, we list our requirements for a user-friendly property specification language for our domain. Fourth, we evaluate eleven visual property specification languages qualitatively and quantitatively, based on our dataset, the running example, and our domain-specific requirements.

The rest of the paper is structured as follows. In Section 2, we introduce the basics of model checking and temporal logic languages. In Section 3, we present the set of properties we have collected from industrial projects, and a running example. In Section 4, we introduce eleven visual property specification languages, utilizing the running example. In Section 5, we then analyze the applicability of the different specification approaches. We discuss our results in Section 6, and present our conclusions in Section 7.

2. Preliminaries

2.1. Model checking

Model checking (Clarke et al., 1999) is a formal verification method that can be used to show that a model of a (hardware or software) system fulfills specified properties. A software tool called model checker is used to automatically determine if a property holds, taking into account all the possible states or executions of the model. If an execution path contradicting to the stated property is found, it is returned to the user as a counterexample.

The desired properties are commonly formalized using temporal logical languages, especially Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) (Clarke et al., 1999).

The challenge to avoid is state space explosion, where the number of model states becomes too big to enumerate through. Symbolic model checkers avoid explicit enumeration through the use of Binary Decision Diagrams (BDD) (Burch et al., 1992). The analysis can also be made faster by using satisfiability (SAT) solvers to perform bounded model checking, where a limit is set for the allowed length of state transition sequences checked (Clarke et al., 2001).

Industrial I&C systems can be based on different platforms, including programmable logic controllers (PLC) using microprocessors, or field-programmable gate arrays (FPGA), or simply hardwired circuits. In digital systems, application logic programming can be based on the standard IEC 61131-3 (IEC, 2013) languages (including function block diagram (FBD)). For FPGAs, the logic can also be specified using hardware description languages. In specialized domains like the nuclear industry, the safety related systems are often built on dedicated platforms, and the vendor-specific programming languages in use do not necessarily follow the IEC standard (Pakonen et al., 2021a).

Model checking has been used to verify industrial I&C logics in various domains, aviation (Menshawy et al., 2018; Stewart et al., 2021), spaceflight (Bozzano et al., 2014), rail traffic (Pavlovic and Ehrich, 2010), automotive industry (Guo et al., 2020; Sharvia and Papadopoulos, 2015; Ljungkrantz et al., 2014), just to name some. In the nuclear domain, in addition to the work we describe in Section 3.1, we can cite, e.g., Yoo et al. (2009), Németh and Bartha (2008), and Adiego et al. (2015).

Symbolic model checkers like NuSMV (Cimatti et al., 2002) are usually based on a discrete model of time. Equating the number of executed transitions with elapsed time, we can get an approximation of the cyclic processing of digital I&C platforms (Ljungkrantz et al., 2010). Instead of perfect realism (e.g., 100 ms per processing cycle), some abstraction is typically needed (Yoo et al., 2009).

If we consider applications where the logic is distributed to different controllers communicating over a network, cyclic time based on a global clock is no longer a realistic assumption (Buzhinsky and Pakonen, 2020). Continuous time model checkers are also available (e.g., UPPAAL (Behrmann et al., 2004)), as well as tools for hybrid systems (e.g., HyComp (Cimatti et al., 2004)). However, in Buzhinsky and Pakonen (2020), where we tried to model networked I&C with timed automata, the resulting computational complexity led to excessive analysis times.

Table 1
Basic temporal operators in LTL.

Op.	Alt.	Semantics
$\mathbf{X}\,p$	$\circ\,p$	("next state"): p is true in the next state of the path.
$\mathbf{G}\,p$	$\Box\,p$	("always" or "globally"): p is true at every state on the path.
$\mathbf{F}\,p$	$\diamond\,p$	("eventually" or "in the future"): p is true at some future state on the path.
$p\,\mathbf{U}\,q$		("until"): q is true at some future state, and at every preceding state on path, p is true.

Table 2
Past temporal operators in LTL.

Op.	Semantics
$\mathbf{Y}\,p$	("yesterday"): p holds in the previous state of the path.
$\mathbf{H}\,p$	("historically"): p is true at every preceding (and the current) state on the path.
$\mathbf{O}\,p$	("once"): p is true at some past (or the current) state on the path.
$p\,\mathbf{S}\,q$	("since"): q is true at some past state, and for every state following it on the path (including the current state), p is true.

2.2. Property formalization

The majority of formal properties can be divided into two types. Safety properties state that something undesirable shall not happen, and a counterexample for such a property is a finite execution path. Liveness properties state that something desirable shall eventually happen (Lamport, 1977), and a finite execution cannot therefore violate such a property—a lasso-shaped (with a loop at the end) counterexample is needed (Clarke et al., 1999).

Temporal logic languages allow the formulation of statements over execution paths rather than individual states (Clarke et al., 1999). LTL and CTL use temporal operators in addition to Boolean operators. The common temporal operators defined for LTL are listed in Table 1.

In addition to specifying future behavior, many interesting properties are more naturally formulated using past temporal operators (Benedetti and Cimatti, 2003), which we list in Table 2.

Throughout this paper, we use p and q to represent Boolean statements over state variables, but in Tables 1 and 2, they can also be nested temporal formulae.

The past operators can be seen as temporal duals of the future operators, **Y** being the dual of **X**, **H** the dual of **G**, **O** the dual of **F**, and **S** the dual of **U** (Benedetti and Cimatti, 2003).

LTL is based on linear time, and CTL on branching time. The CTL operators are obtained by adding the path quantifiers **E** ("for some execution path") and **A** ("for all execution paths"), e.g. **EX** or **AX**.

Property Specification Language (PSL) (IEC, 2012) is an extension of LTL and CTL, designed to be compatible with hardware description

languages. The aim is that the specifications are "human readable" (Eisner and Fisman, 2006). PSL uses two styles: LTL style and Sequential Regular Expressions (SERE) style.

The LTL style provides syntactic sugaring (`always` for **G**, `never` for **G**¬, etc.), but also useful operator variations. A variant of the `next` operator allows one to conveniently state the property "`a` leads to `b` for the following three to five cycles" as (Eisner and Fisman, 2006):

```
always (a -> next_a[3:5](b))
```

The SERE style is used to describe multi-cycle behavior as a series of Boolean expressions. A SERE is enclosed in curly braces, and the "atoms" of the SERE are separated by semicolons. As an example (Eisner and Fisman, 2006):

```
always {req} |=> {ack;busy[*3];done}!
```

Using the implication operator `|=>` and the repetition operator `[*n]`, the above formula translates to: "if `req` is true, then, starting at the next cycle, there must follow a sequence where `ack` is true for one cycle, then `busy` is true for three cycles, and then `done` is true for one cycle". The equivalent nested LTL property would be harder to write and read.

In specifying functional requirements (and therefore also verifiable properties) for industrial control systems, there is a specific need to address sequences in time, mostly due to the dynamic behavior of the physical processes being controlled, and feedback from both the processes and human users. Desired effects of taken actions do not occur immediately, but take place within different time windows. Proper timing is often crucial for system design, be it a matter of simple delays or execution of more complex sequences. The need to address timing can be due to different factors:

- Filtering of sporadic signals (due to, e.g., electromagnetic interference, flickering switches, or measurements fluctuating near the limit threshold) may require that signals are only acknowledged after having been active for some time (on-delay).
- Actuation commands may need to be held (off-delay) for a given time to ensure that actuators have sufficient time to reach a desired state.
- Complex sequences may need to be executed in transient situations, such as process start up. Controlling actions will need to take place in a particular order, to minimize production losses, save energy, or protect the equipment.
- Slow response of the controlled processes has to be accounted for. For example, a tank full of liquid is not heated in an instant. It also takes a long while for process equipment such as large generators or motors to fully start or stop.
- Feedback from human users has to be taken into account. Consider, for example, an application where a low-level alarm will not directly lead to actions, if an operator acknowledges the alarm within a given time limit.

Languages such as LTL and CTL are more concerned with the relative order of events than explicit time, although, as mentioned above, discrete time can still be used to model the cyclic processing of digital I&C. The LTL operators **X** and **Y** then allow the expression of discrete-time properties, and PSL makes it easier to describe multi-cycle behavior. For more exact handling of time, real-time specification languages such as MTL and TCTL were developed (see, e.g., Boyer (2009)). However, specification of real-time properties is outside the scope of this study, as analysis of networked I&C systems with timed automata has proven too computationally costly (Buzhinsky and Pakonen, 2020).

2.3. Property patterns

A common solution to bridge the gap between generally acceptable, natural language requirements and formal specification languages is to use property specification patterns. A specification pattern describes some aspect of a system's desired behavior and provides suitable expressions in different formalisms.

A particularly oft-cited collection of patterns was proposed in Dwyer et al. (1998, 1999), where Dwyer et al. introduce a system of patterns for finite-state verification tools such as model checkers, organized into hierarchies to support browsing. A pattern consists of a name, a statement of the pattern's intent, mappings into formal logic, examples of pattern use, and relationships to other patterns. Each pattern has a scope that specifies the extent of the model execution over which the pattern must hold (Globally, Before R, After Q, Between Q and R, After Q until R).

2.4. Related research

In addition to — and inspired by — the work of Dwyer et al. different researchers have proposed their own property specification patterns:

- In Campos et al. (2008), Campos and Machado (2009), the authors propose a collection of domain-specific LTL and CTL specification patterns for automated production systems, based on analysis of different case studies collected from literature.
- In Menghi et al. (2021), the authors propose a set of LTL and CTL patterns for verifying the mission behavior of mobile robots, by analyzing natural language requirements found in literature.
- In Monteiro et al. (2008), the authors propose a set of CTL patterns for determining properties of cellular interaction networks, based on a list of questions the authors collected from different modeling studies in systems biology.
- In Zhang et al. (2021), the authors propose a method for mining Propositional Projection Temporal Logic (PPTL) properties based on a pattern library. The pattern library was built based on a literature survey, and contains some common properties that can be expressed in CTL and LTL.

The difference of the above-mentioned work to our research is that our analysis is based on actual formal properties from practical model-checking projects. As far as we are aware, no similar dataset has been collected in the I&C domain. In Ljungkrantz et al. (2014), the authors collected about one hundred example properties from car and aerospace component manufacturing industry, but the properties were not formal to begin with, but formalized by the authors for research purposes.

In our previous work (Buzhinsky, 2018), we identified patterns in formal properties we specified for a research study on closed-loop modeling (Preuße et al., 2012), where the feedback from the controlled environment (in Buzhinsky (2018), a nuclear power plant) is included in the model. Four of the patterns do not appear in our data, which is based on open-loop modeling (only the I&C logic is modeled).

Evaluation of user-friendly property specification approaches is a topic also addressed in other works:

- In Filipovikj et al. (2016), the authors present a tool called SESAMM Specifier, which supports different property specification notations (including RTGIL (Ramakrishna et al., 1996) and L_R (Lee and Sokolsky, 1997)—see Section 4). The authors asked requirements engineering experts to select the most intuitive approach, and found that each of the six suggested notations was preferred by at least one expert.
- In Grobelna (2020), the author conducted an experiment where university students learned about LTL, CTL, and a graphical approach called Scratch (see Section 4), with most students preferring Scratch.
- In Czepa and Zdun (2020), the authors experiment with the understandability of different representations, and report that computer science students prefer the patterns from Dwyer et al. over LTL.

In addition to our specific focus on the I&C domain, the difference to our work is that we are working with real formal properties, written by experts having worked on practical model-checking projects for years. We have also considered a wider range of languages than the above-mentioned works.

An ideal solution would be automatic translation of natural language requirements into formal specifications. In Brunello et al. (2019), Brunello et al. present a critical review of works on translating English to LTL, suggest possible research directions (based on, e.g., neural networks, or evolutionary algorithms), and conclude their review by stating that a general solution is still missing.

To mention a few works on translation in the I&C domain:

- In Santos et al. (2018), the authors use controlled natural language to translate statements about a system's environment to restrictions for a formal system model. The objective is to create a closed-loop model, and thereby reduce the model state space. One of the case studies is a safety system from a nuclear power plant. The LTL mapping is limited to four templates and three scopes.
- In Nikora and Balcom (2009), the authors use machine learning and natural language processing techniques to identify the patterns of Dwyer et al. in natural language requirements. Their case study dataset is based on requirements for space mission systems. The focus is on identification, translation is only mentioned as a future work topic.
- In Konrad and Cheng (2005), the authors extend the patterns from Dwyer et al. to support real-time property specification. They specify a mapping from structured English to MTL, TCTL, and RTGIL, demonstrated with an automotive embedded system. In Autili et al. (2015), the authors build upon that work to construct a framework and a tool for specifying real-time and probabilistic properties. In their tool, the user can express properties in structured English, and given specifications can then automatically be translated to, e.g., MTL, based on catalogue of 40 real-time patterns.
- In Giannakopoulou et al. (2020), the authors use a structured natural language (combining features from the Dwyer et al. patterns and Easy Approach to Requirements Syntax (EARS) templates (Mavin et al., 2009)) and RTGIL templates to construct metric temporal logic (MTL) formulae. Their case study involves an aircraft control system.

3. I&C logic properties

3.1. Industry projects

Since 2008, VTT has been applying model checking in practical customer projects in the Finnish nuclear (Pakonen, 2021a,b) and railway industries. The projects include:

- On commission from the Finnish Radiation and Nuclear Safety Authority (STUK), VTT has analyzed the detailed design of safety-classified I&C systems of the new-build Olkiluoto 3 nuclear power plant (NPP).
- For the utility Fennovoima, VTT has verified the early, functional design for I&C intended for the (since canceled) Hanhikivi-1 NPP new-build project.
- For the utility Fortum, VTT has verified the detailed design of several systems in the LARA and ELSA I&C renewal projects for the Loviisa NPP.
- For the utility TVO, VTT has analyzed the early functional design and the I&C logic design for the I&C renewal project for the Olkiluoto 1&2 NPPs.
- For the engineering company Mipro, VTT has analyzed the design of I&C software of railway point control and route setting logics.

Table 3
The use of temporal operators in the LTL properties.

Op.	No. of properties	%
G	2971	99.6
X	570	19.1
Y	141	4.73
O	132	4.42
F	52	1.74
H	11	0.37
S	11	0.37
U	8	0.27

The basis for the verification work has alternated between standard IEC 61131-3 function block diagrams, non-standard vendor-specific block diagrams, diagrams drawn with MS Office tools, and VHDL code. The platforms for the verified logics have included PLCs, FPGAs, and hardwired circuits.

In all the practical projects combined, VTT has to date detected a total number of 109 design issues, in many cases leading to redesign. The safety relevance and/or the probability of the issues varies significantly. Some of the issues may not be relevant when considering the broader context of the analyzed system.

To evaluate the applicability of different approaches on user-friendly property specification, we have collected properties written and verified by VTT analysts in the above-mentioned projects between the years 2014 and 2022. The properties have been written by two analysts:

- Analyst A has a degree in I&C software engineering. They have worked on the practical projects since 2007. They are responsible for the majority (92%) of the collected properties.
- Analyst B has a degree in computer science. They have worked on the practical projects between 2008 and 2016, and have written 8% of the collected properties.

The properties have been written in LTL, CTL and PSL, to the extent in which PSL is supported by NuSMV.

The verification of function block diagrams has been performed using the graphical modeling tool MODCHK (Pakonen et al., 2021a), a front-end for NuSMV, developed by VTT and Fortum. The work process is described in Pakonen et al. (2021a).

3.2. The sample data

Our dataset (Pakonen, 2023) contains 3923 properties. 2984 (76%) are in LTL, and there are 122 unique types in terms of their temporal structure. 561 (14%) are in PSL, with 39 unique structures. 378 (10%) are in CTL, with only 5 unique structures (with 373 instances of "**AG EF** *p*").

For all the LTL properties written, the occurrence of different temporal operators is shown in Table 3.

Nearly all of the PSL properties use SEREs, and specifically, 514 (92%) of the PSL properties contain at least one SERE that employs the repetition operator `[*n]`.

3057 (78%) of all properties can be classified as implications, due to the use of the operator →, or the PSL SERE suffix implication operators `|->` ("at the same cycle as the cycle in which the left-hand side finishes") and `|=>` ("the cycle after the cycle in which the left-hand-side finishes") (Eisner and Fisman, 2006).

Regarding timing, of all properties, 1257 (32%) refer to a specific future or past state (by using the LTL operator **X** or **Y**, or PSL SEREs).

Out of the sample data, we can identify patterns (types of properties) that repeat. In Table 4, we list the property types that are used at least three times, and appear in the context of more than one model (I&C function or system).

In Table 5, we separately list the types of properties that revealed actual design issues. Here, we also include the types we omitted from Table 4.

Table 4
Recurring types of properties (overall).

#	Generalized property type	Count	%
1	$\mathbf{G}\ p$	2033	52
2	$\mathbf{G}(p \rightarrow \mathbf{X}\ q)$	445	11
3	`always` {*SERE*} `|->` {*SERE*}`!`	415	11
4	$\mathbf{AG}\ \mathbf{EF}\ p$	373	9.5
5	`never` {*SERE*}	124	3.2
6	$\mathbf{G}(p \rightarrow \mathbf{O}\ q)$	105	2.7
7	$\mathbf{G}\ e \rightarrow \mathbf{G}\ p$	65	1.7
8	$\mathbf{G}(p \wedge \mathbf{X}\ q \rightarrow \mathbf{X}\,\mathbf{X}\ r)$	40	1.0
9	$\mathbf{G}(p \rightarrow \mathbf{F}\ q)$	31	0.8
10	$\mathbf{G}(p \wedge \mathbf{X}\ q \rightarrow r)$	24	0.6
11	$\mathbf{G}(p \rightarrow \mathbf{Y}\ q)$	20	0.5
12	`always` {*SERE*} `|=>` {*SERE*}`!`	19	0.5
13	$\mathbf{G}(p \wedge \mathbf{X}\ q \rightarrow r \wedge \mathbf{X}\ s)$	12	0.3
14	$\mathbf{G}(p \rightarrow (\neg(\mathbf{Y}\ p \wedge \neg p)\ \mathbf{S}\ q))$	11	0.3
15	$i \wedge \mathbf{G}\ e \rightarrow \mathbf{G}\ p$	10	0.3
16	$\mathbf{G}\ e \rightarrow \mathbf{G}(p \rightarrow \mathbf{X}\ q)$	10	0.3
17	$\mathbf{G}(p \wedge \mathbf{X}\ q \rightarrow \mathbf{X}\ r \wedge \mathbf{X}\,\mathbf{X}\ s)$	8	0.2
18	i	6	0.1
19	$i \rightarrow \mathbf{G}\ p$	6	0.1
20	$i \rightarrow \mathbf{G}(p \rightarrow \mathbf{O}\ q)$	5	0.1
21	$\mathbf{G}(p \wedge \mathbf{X}\ q \wedge \mathbf{X}\,\mathbf{X}\ r \rightarrow \mathbf{X}\,\mathbf{X}\,\mathbf{X}\ s)$	5	0.1
22	$i \wedge \mathbf{G}(q \rightarrow \mathbf{X}\ r) \rightarrow \mathbf{G}\ p$	4	0.1
23	$\mathbf{G}(q \rightarrow \mathbf{X}\ r) \rightarrow \mathbf{G}\ p$	3	0.1
	Other	149	3.9

Table 5
Types of properties that have revealed design issues in practical projects.

#	Generalized property type	Issues
1	$\mathbf{G}\ p$	27[b]
2	$\mathbf{G}(p \rightarrow \mathbf{X}\ q)$	13[c]
3	`never` {*SERE*}	11
4	`always` {*SERE*} `|->` {*SERE*}`!`	9
5	$\mathbf{G}\ e \rightarrow \mathbf{G}\ p$	8[d]
6	$\mathbf{G}(p \rightarrow \mathbf{O}\ q)$	5
7	$\mathbf{AG}\ \mathbf{EF}\ p$	2
8[a]	$\mathbf{G}\ e \rightarrow \mathbf{G}(p \rightarrow \mathbf{O}\ q)$	2
9	$\mathbf{G}\ e \rightarrow \mathbf{G}(p \rightarrow \mathbf{X}\ q)$	2
10	$i \wedge \mathbf{G}\ e \rightarrow \mathbf{G}\ p$	1
11	$i \rightarrow \mathbf{G}(p \rightarrow \mathbf{O}\ q)$	1
12[a]	$\mathbf{G}\ e \rightarrow \mathbf{G}(p \rightarrow \mathbf{F}\ q)$	1
13[a]	$\mathbf{G}\ e \rightarrow \mathbf{G}\neg(p \wedge \mathbf{X}\ q \wedge \mathbf{X}\,\mathbf{X}\ r)$	1
14[a]	$i \rightarrow \mathbf{G}(p \wedge \mathbf{X}\ q)$	1
15[a]	$\mathbf{G}(p \rightarrow (q \vee \mathbf{O}\ r))$	1
16[a]	$\mathbf{G}\neg(p \wedge \mathbf{X}\ q \wedge \mathbf{X}\,\mathbf{X}\ r)$	1
17[a]	$\mathbf{G}((p \wedge \mathbf{X}\ q) \rightarrow (\mathbf{X}\ r \vee s \vee \mathbf{Y}\ s \vee \mathbf{Y}\,\mathbf{Y}\ s))$	1
18[a]	$\mathbf{G}(p \rightarrow \mathbf{O}(\neg q \wedge \mathbf{Y}\ q))$	1
19[a]	$\mathbf{G}(p \rightarrow \mathbf{O}(q \vee (r \wedge \mathbf{O}\ s)))$	1
20[a]	$i \wedge \mathbf{G}\neg(((\neg q \wedge \mathbf{X}\ q) \vee (\neg r \wedge \mathbf{X}\ r)) \vee ((\neg s \wedge \mathbf{X}\ s) \vee (\neg t \wedge \mathbf{X}\ t)) \rightarrow \mathbf{G}\ p$	1

[a] Property is not listed in Table 4.
[b] See examples in Pakonen and Björkman (2017), Pakonen (2021b).
[c] See examples in Pakonen (2021a,b), Pakonen et al. (2021b).
[d] See an example in Pakonen et al. (2021a).

In Tables 4 and 5, we have generalized some of the properties from the form they originally appeared by combining the ones whose temporal formulae are equivalent. As an example, as seen in Table 6, $\mathbf{G}(a \wedge \mathbf{X}\ b \rightarrow \mathbf{X}\ c)$ is a specialized version of $\mathbf{G}(p \rightarrow \mathbf{X}\ q)$ if we substitute p for a and q for $(b \rightarrow c)$. The most natural form may depend on personal preference, and the type of property being specified. For instance, if we consider a property: "a rising edge of the signal *trig* shall lead to *act*", it may be more natural for some analyst to specify $\mathbf{G}(\neg trig \wedge \mathbf{X} trig \rightarrow \mathbf{X} act)$ rather than $\mathbf{G}(\neg trig \rightarrow \mathbf{X}(trig \rightarrow act))$, which could be one reason for this variant being so common (317 instances) in our data.

Some patterns can also be transformed to each other by simple Boolean transformations. For instance, $\mathbf{G}\ p$ and $\mathbf{G}\ \neg p$ are equivalent in the temporal sense, so we have only included $\mathbf{G}\ p$ here.

In Tables 4 and 5, we also do not make a distinction between the different temporal structures in the SEREs within the PSL types.

Table 6
Temporally equivalent variants of $\mathbf{G}(p \rightarrow \mathbf{X}\ q)$ appearing in our data.

Variant	p	q	Count	%
$\mathbf{G}(a \wedge \mathbf{X}\ b \rightarrow \mathbf{X}\ c)$	a	$(b \rightarrow c)$	317	8.1
$\mathbf{G}(\mathbf{Y}\ a \wedge b \rightarrow c)$	a	$(b \rightarrow c)$	52	1.3
$\mathbf{G}\neg(a \wedge \mathbf{X}\ b)$	a	$\neg b$	34	0.9
$\mathbf{G}(a \rightarrow \mathbf{X}\ b)$	a	b	28	0.7
$\mathbf{G}\neg(\mathbf{Y}\ a \wedge b)$	a	$\neg b$	8	0.2
$\mathbf{G}(\mathbf{Y}\ a \rightarrow b)$	a	b	6	0.2

In our online dataset, the reader can find both the original properties exactly as the analysts specified them, and their more general forms we have listed here.

In both Tables 4 and 5, we have used blue color to highlight cases where the analyst has apparently excluded some scenarios, by

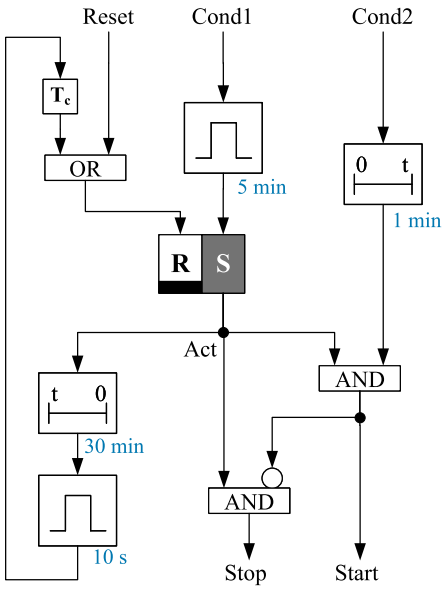

Fig. 1. Logic for our running example.

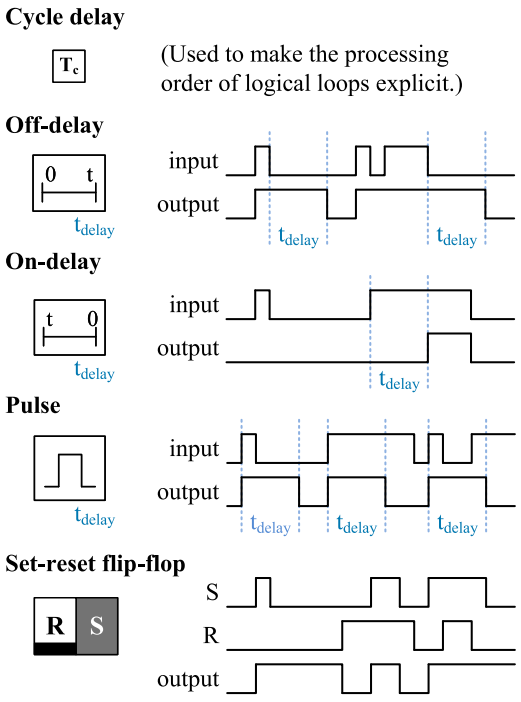

Fig. 2. Key for the blocks used in the example.

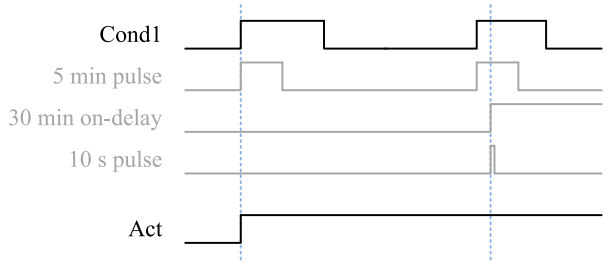

Fig. 3. Counterexample for p_4.

specifying that initially, *i* shall hold, and/or globally, *e* (or some temporal property) shall hold. The reasons for the analyst to add such limitations include:

1. filtering out irrelevant execution paths, e.g., input combinations not possible in the model environment (alternatively, such constraints could also be included in the model),
2. checking to see if alternative (e.g., more probable) counterexamples are possible, or
3. wishing to understand how the system (model) behaves under certain assumptions.

The reason for the CTL property type "**AG EF** *p*" occurring so often is that it can be used as a quick sanity check, or to verify that the signal *p* cannot be stuck into its false value, i.e., permanently freeze (to false). Campos and Machado (2009) includes this type as the pattern "Possibility", and Campos et al. (2008) as the type "Occurrence".

3.3. Running example

As a running example, let us consider a safety function shown in Fig. 1. The function contains two masked and simplified examples of real-world nuclear I&C design issues detected in practical projects. We have combined logics that were originally intended for different functions. We only include the function blocks that are needed to reproduce the underlying design issues. We have modified the graphical block symbols, variable names, and the delay lengths to mask the origins. The processing logic for the basic blocks is described in Fig. 2.

The safety function is initiated (Act) when the process monitoring condition Cond1 (e.g., high pressure or temperature) activates. The order to the controlled device depends upon the state of another condition—if Cond2 holds, the device is started, otherwise it is stopped. After five minutes, the operator can reset the function manually. Otherwise, after thirty minutes, the function is reset automatically. In the absence of any commands from the safety function, normal operation I&C can then continue operating the device.

The properties specified for the safety function are listed in Table 7. They have been written based on natural language functional requirement specifications for the original system. The properties appear here in the form they have taken after the analysis work. Since model checking can also reveal an error in a property, some iteration might have been needed to ensure they are correct. We discuss potential threats to correctness and completeness in Pakonen et al. (2021a).

To illustrate the power of model checking, and for the benefit of readers interested in I&C design issues, we include the detected counterexamples below. For p_4, NuSMV produced the counterexample shown in Fig. 3, where Cond1 is reactivated just as the 30-min delay is about to pass, which means that the 5-min pulse will prevent the 10-s pulse from automatically resetting the flip-flop. As a result, the safety function remains active longer than allowed.

Also, p_7 is proven false by the counterexample shown in Fig. 4, where Cond2 is reactivated just as the 5-min pulse is about to end, and the operator then immediately resets the function. As a result, the timing of the operator action prevents the 1-min off-delay for Cond2 from actually keeping the (second) Start command active long enough for the device to start.

4. Visual specification languages

In this section, we introduce eleven visual property specification languages, utilizing our running example from Section 3.3.

4.1. Constraint diagram

Introduced year: 1996
Original domain: real-time reactive systems

Table 7
Properties for our running example.

	Lang.	Property	Functional requirement
p_1	LTL	$\mathbf{G}((\neg\text{Cond1} \wedge \neg\text{Act} \wedge \mathbf{X}\,\text{Cond1}) \rightarrow \mathbf{X}\,\text{Act})$[a]	Activation of Cond1 shall lead to Act.
p_2	PSL	`always {(Act & !Cond1)[*10];(Reset & !Cond1)} |-> {!Act}!`[b]	After 5 min, an the operator can reset Act.
p_3	PSL	`always {!Act;Act} |-> {Act[*10]}!`[b]	The operator cannot reset Act1 before 5 min have passed.
p_4	PSL	`never {Act[*22]}`[b]	Act shall be automatically reset after 30 min.
p_5	LTL	$\mathbf{G}((\neg\text{Act} \wedge \mathbf{X}\,\text{Act}) \rightarrow (\neg\text{Cond1} \wedge \mathbf{X}\,\text{Cond1}))$	Act shall never be actuated spuriously.
p_6	LTL	$\mathbf{G}((\text{Act} \wedge \text{Cond2}) \rightarrow \text{Start})$	If Cond2 is TRUE, activation of the function shall set Start.
p_7	PSL	`always {Start & Cond2);!Cond2[*3]} |-> {Start}!`[b]	Reset of Start shall always be delayed for one minute.
p_8	PSL	`always {(Act & !Cond2)[*4]} |-> {Stop}!`[b]	If Cond2 is FALSE, activation of the function shall set Stop.
p_9	LTL	$\mathbf{G}\neg(\text{Start} \wedge \text{Stop})$	The system shall not set contradictory commands.
p_{10}	LTL	$\mathbf{G}(\text{Start} \rightarrow (\text{Act} \wedge (\text{Cond2} \vee \mathbf{Y}\,\text{Cond2} \vee \mathbf{Y}\,\mathbf{Y}\,\text{Cond2} \vee \mathbf{Y}\,\mathbf{Y}\,\mathbf{Y}\,\text{Cond2})))$	Start shall never be actuated spuriously.
p_{11}	LTL	$\mathbf{G}(\text{Stop} \rightarrow (\neg\text{Cond2} \wedge \text{Act}))$	Stop shall never be actuated spuriously.
p_{12}	CTL	$\mathbf{AG}\,\mathbf{EF}\,\text{Start}$	Start shall not permanently freeze to FALSE.
p_{13}	CTL	$\mathbf{AG}\,\mathbf{EF}\,\neg\text{Start}$	Start shall not permanently freeze to TRUE.
		...[c]	...

[a] $\mathbf{G}(\text{Cond} \rightarrow \text{Act1})$ would be FALSE because of the 5-min pulse (and see p_2).
[b] Here, we assume that 4 time steps correspond to 1 min, 10 steps to 5 min, and 20 steps to 30 min.
[c] The $\mathbf{AG}\,\mathbf{EF}$ patterns of p_{12} and p_{13} repeat for all relevant variables.

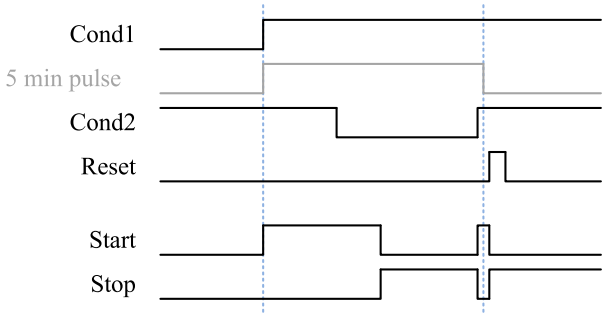

Fig. 4. Counterexample for p_7.

Case example: gas burner
Inspired by: timing diagrams

Constraint Diagrams (Dietz, 1996) provide a graphical notation for assumption/commitment requirements for real-time systems. They consist of waveforms that describe state changes over Boolean expressions. The state change point is called an event. Timing requirements between events are marked with an arrow labeled with a time interval. Time intervals can also be specified for states on the waveform. A dashed waveform line indicates unconstrained behavior.

The commitments are marked with boxes. When the assumption waveforms are observed, the system shall also fulfill the commitment waveform and arrows.

In Fig. 5, we show the properties p_3, p_4, p_5, p_8 and p_9. In Fig. 16, we also show the Constraint Diagram for $\mathbf{G}(p \rightarrow \mathbf{O}\,q)$ to highlight the fact that we can also specify commitments to past behavior, here by marking the commitment box to the q event.

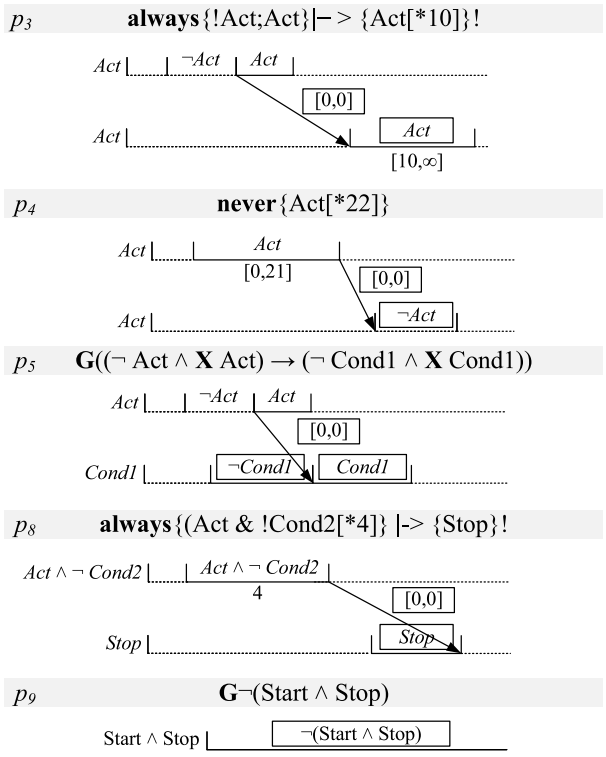

Fig. 5. p_3, p_4, p_5, p_8, and p_9 specified using Constraint Diagrams.

4.2. GIL

Introduced year: 1994
Original domain: concurrent software systems
Case example: elevator
Inspired by: timing diagrams

In Graphical Interval Logic (GIL) (Dillon et al., 1994), properties of legal state sequences are expressed with timing diagrams. An interval (denoted by a left-closed right-open line segment) specifies the context within which properties hold. Search primitives (dashed arrow-ended line) locate the first future point at which their target holds. A triangle is used as a point operator, locating a point, and asserting that a property holds over the interval specified below, starting at that point. Layout determines the grouping of operations.

In see Fig. 6, we show the properties p_6 and p_9, and the liveness property $\mathbf{G}(q \rightarrow \mathbf{F}\,q)$ for illustration purposes. We also include $\mathbf{G}(\neg p \wedge \mathbf{X}\,p \rightarrow \mathbf{X}\,q)$ to clarify a point we make in Section 5.1 below.

A real-time extension of GIL (RTGIL) is introduced in Ramakrishna et al. (1996).

4.3. TimeLine editor

Introduced year: 2001
Original domain: software systems
Case example: Internet server
Inspired by: timing diagrams

The TimeLine Editor (Smith et al., 2001) uses a representation of the timeline as a horizontal bar, with vertical marks notating interesting event occurrences as they are generated over time. The system events

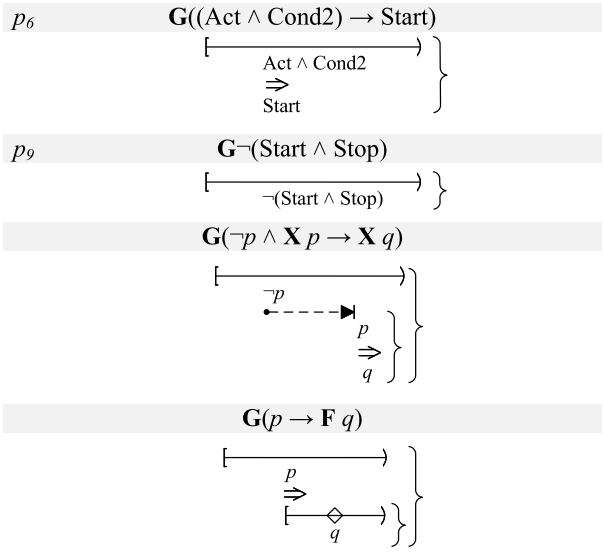

Fig. 6. p_6, p_9, $\mathbf{G}(\neg p \wedge \mathbf{X}\, p \rightarrow \mathbf{X}\, q)$ and $\mathbf{G}(q \rightarrow \mathbf{F}\, q)$ specified using GIL.

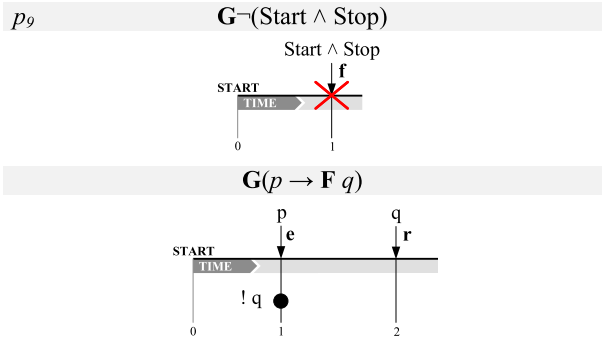

Fig. 7. p_9 and $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$ specified using TimeLine Editor.

come in three types: "e" for regular events (that do not always need to occur but make a property relevant when they do), "r" for required events (that must occur in response to regular events), and "f" for fail evens (that should not occur). Actual timing is not addressed (even in approximate terms), only the ordering of events. Constraint, marked with horizontal lines, specify the absence of particular events over certain intervals. A constraint either includes the begin/end points of the interval (line starts/ends with a black circle) or excludes them (line starts/ends with a black bar next to the begin/end point).

In Fig. 7, we show p_9 expressed as a fail event, along with the liveness property $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$.

4.4. TDE

Introduced year: 2007
Original domain: industrial I&C
Case example: PLC logic
Inspired by: timing diagrams

A visual language developed for a tool called Timing Diagram Editor (TDE) is introduced in Vyatkin and Bouzon (1994). The left part of the diagram specifies the "precondition" — initial signal level states or transitions — whose occurrence leads to behavior accepted by the diagram. Signals can have the values "zero", "any" (can arbitrarily change), "stable" (arbitrary, but does not change) or "one". Signal changes within the precondition are assumed simultaneous, but otherwise, partial ordering is used, which means that that ordering between

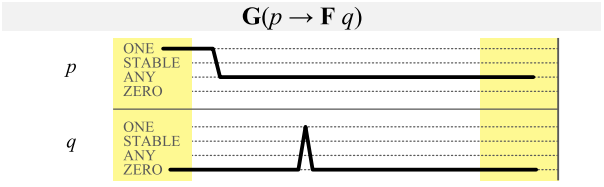

Fig. 8. $\mathbf{G}(q \rightarrow \mathbf{F}\, q)$ specified using TDE.

events is not determined by their horizontal position on the time line. In order to constrain ordering, precedence operators are used.

For illustration purposes, we show a simple example specifying the property $\mathbf{G}(q \rightarrow \mathbf{F}\, q)$ in Fig. 8. Since there are no precedence operators, the change of p from true to "any" can occur before, during, or after q being true.

4.5. STD

Introduced year: 1993
Original domain: reactive systems
Case example: industrial I&C platform
Inspired by: timing diagrams

Another language similar to TDE is Symbolic Timing Diagrams (STD) (Schlör, 2002). In STD, diagrams also consist of waveforms, each waveform describing the value of a Boolean expression over time. Falling and rising edges of waveforms are called events. Progress of the timeline is restricted by constraints. The activation condition for the diagram is the value of model signals matching the associated values at the beginning of each waveform.

In Fig. 9, we show our exemplar properties p_1, p_3, p_4, p_8, and p_9, expressed using variants of STD. (We are unsure if our constructs for p_3 and p_4 are legal.) With p_1 p_3, p_4 and p_8, we have used a distance measure (line with arrows) to define distance between events in terms of model steps. This constraint type is not formally defined in Schlör (2002), but is used in an extension of STD called STDx. The dashed line in p_8 expresses a weak constraint, which "deactivates" the diagram if the criterion (Act & !Cond2) resets.

4.6. PSC

Introduced year: 2007
Original domain: message exchange in concurrent systems
Case example: ship onboard communication system
Inspired by: UML Sequence diagram

Property Sequence Chart (PSC) (Autili et al., 2007) is a scenario-based language for expressing temporal properties. PSC extends a subset of UML 2.0 Interaction Sequence Diagrams, introducing graphical elements that specify ordering and constraints for messages passed between communicating system components. Some ideas, terms, and graphical elements are borrowed from TimeLine Editor. In PSC diagrams, time runs from the top down. Messages fall in three categories familiar from TimeLine Editor, with the prefix "e" for regular, "r" for mandatory, and "f" for fail messages.

In Fig. 10, we show our exemplar properties p_1, p_2, p_4, and p_9. The fail messages allow us to specify safety properties. The "strict operator" (black bar on the vertical line) serves as the **X** operator, and the "loop operator" makes the specification of PSL properties easier.

A real-time extension to PSC called Timed PSC is introduced in Zhang et al. (2010), and a related real-time approach called Modal Sequence Diagrams is introduced in Fockel et al. (2018).

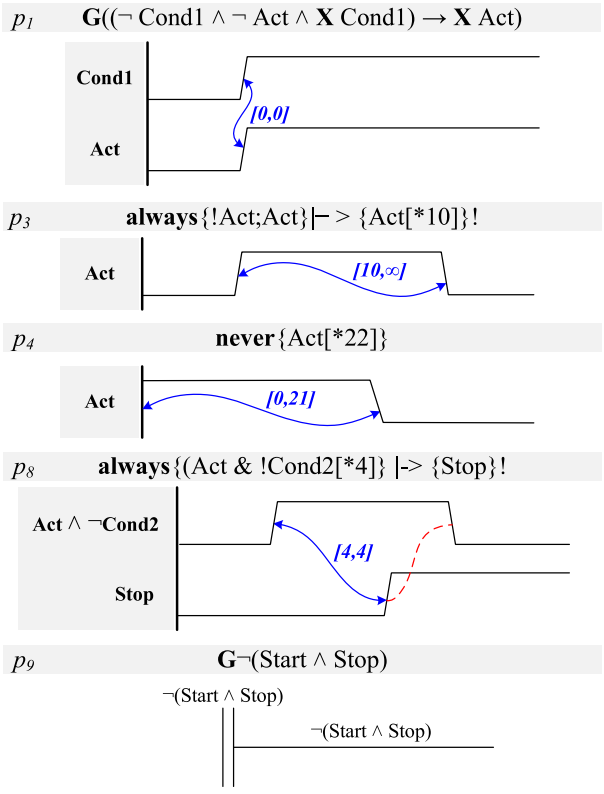

Fig. 9. p_1, p_3, p_4, p_8 and p_9 specified using variants of STD.

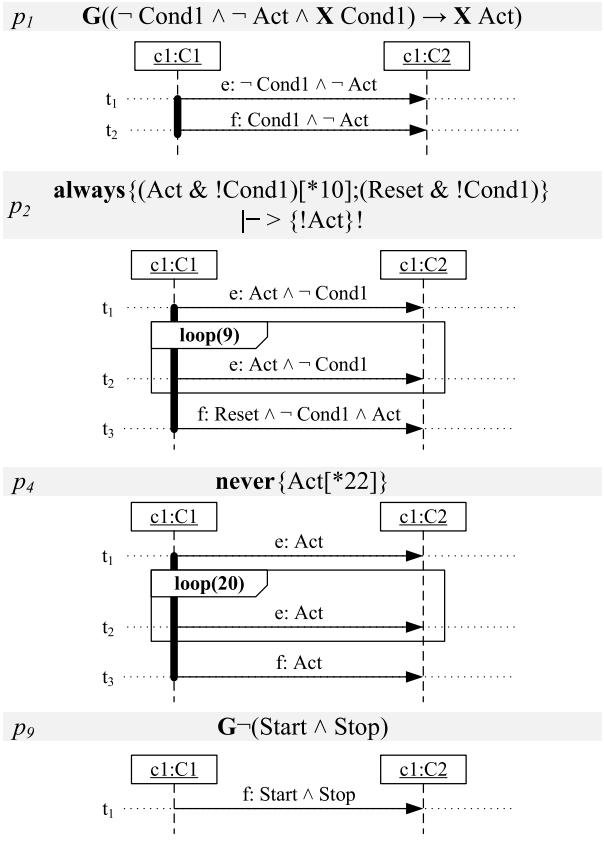

Fig. 10. p_1, p_2, p_4, and p_9 specified using PSC.

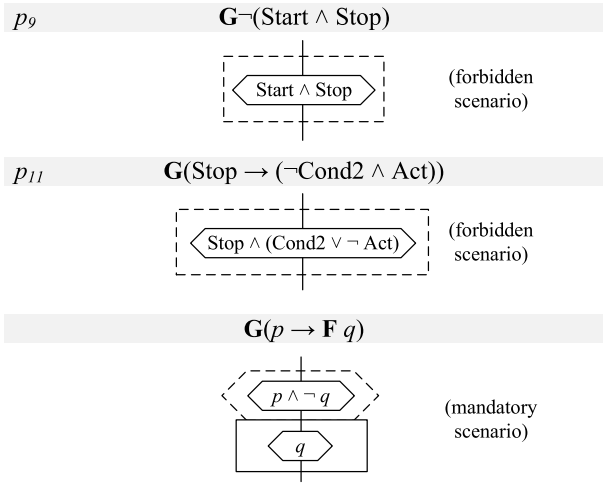

Fig. 11. p_9, p_{11}, and $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$, specified using LSC.

4.7. LSC

Introduced year: 2001
Original domain: message exchange
Case example: automated rail-car system
Inspired by: UML Sequence diagram

Live Sequence Charts (LSC) (Damm and Harel, 2001) represent another attempt of modifying UML charts in order to capture properties. LSC scenarios are described in terms of charts, locations, messages, and conditions, each of which can be "hot" (mandatory) or "cold" (provisional), with the latter type represented with dashed lines.

In Fig. 11, we show properties p_9 and p_{11}, and — to illustrate a liveness property — $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$. For p_9 and p_{11}, we use an "existential chart" to specify that if a condition is eventually reached, a forbidden scenario has taken place. For $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$, we first specify a cold "pre-chart" which, if traversed successfully ($p \wedge \neg q$ is reached), triggers the following hot chart, where we expect q to eventually occur.

4.8. L_R

Introduced year: 1997
Original domain: real-time systems
Case example: telephone answering machine
Inspired by: CTL

In L_R (Lee and Sokolsky, 1997), properties are visualized as directed acyclic graphs. Nodes of the graph represent predicates, logical connectives (and, or, not), modal operators ("next state" properties referring to communication events), and temporal operators. Path quantifiers ($\forall$, $\exists$) are also used. Notably, the temporal operators are defined by a formal method expert for each application domain. A keyword such as "eventually" or "always" is used in the graphs, whereas the matching formal template is specified manually.

We show one example of how our exemplar properties p_1 and p_7 could be expressed in Fig. 12. We also include p_{12}, one of our two CTL properties.

4.9. Scratch

Introduced year: 2020
Original domain: I&C systems
Case example: smart home system
Inspired by: LEGO bricks

In Grobelna (2020), Grobelna suggests the use of Scratch (Resnick et al., 2009) for specifying LTL properties. Developed as an easily

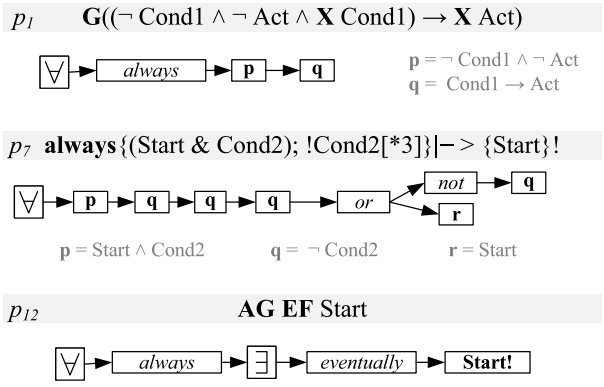

Fig. 12. p_1, p_7 and p_{12} specified using L_R.

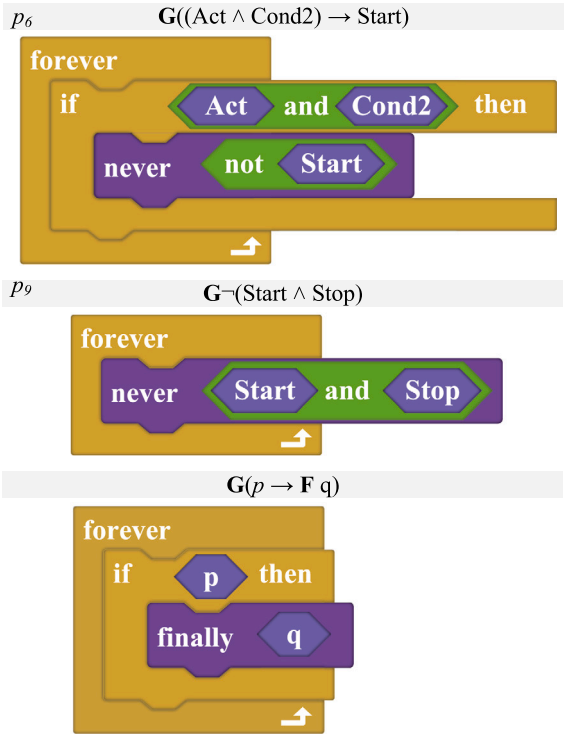

Fig. 13. p_6, p_9 and $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$ specified using Scratch.

approachable visual language for programming games and animations (Resnick et al., 2009), Scratch is based on colorful, graphical blocks that "snap" together.

In Fig. 13, we show p_6 and p_9 (along with $\mathbf{G}(q \rightarrow \mathbf{F}q)$, for illustration purposes). As the "if then" conditional block only accepts formulae beginning with either **X**, **F** or ¬ on the "then" side, p_6 needs to be stated as "never not".

4.10. Simulink design verifier

Introduced year: 2007 (SLAC National Accelerator Laboratory, 2007)
Original domain: discrete time systems
Case example: various Simulink models
Inspired by: function block diagrams

Simulink Design Verifier (Mathworks, 2023) enables modeling and proving safety requirements for discrete time systems. Properties are specified using function block diagrams. Temporal properties are expressed using special function blocks, including Detector (for detecting

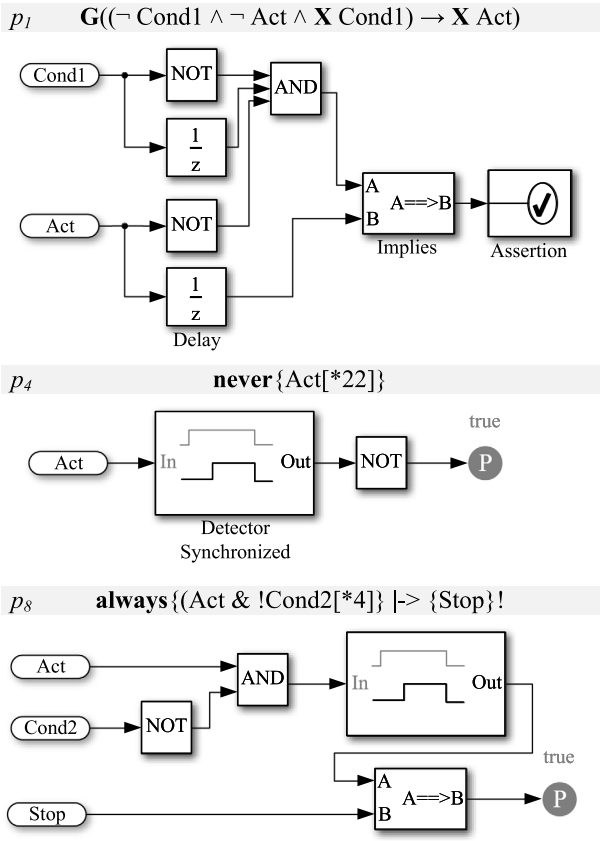

Fig. 14. p_1, p_4 and p_8 specified using Simulink Design Verifier.

and constructing signals with a user-specified length of true duration), Extender (extending the true duration of an input signal by a fixed number of steps or indefinitely), and Within Implies (observing whether the second input is true for at least one step within each true duration of the first input).

In Fig. 14, we use the Implies and Detector blocks to specify p_1, p_4 and p_8. (The output of the "Synchronized" variant of Detector stays true as long as the input continues to be true.)

4.11. VTS

Introduced year: 2004
Original domain: real-time systems
Case example: remote sensing
Inspired by: event scenarios

VTS (Alfonso et al., 2004) is a scenario-based notation aimed at expressing real-time requirements. The notation is based on points connected by lines and arrows. Points stand for sets of events. An arrow between two points defines the precedence of the events. The event labels above the arrows stand for forbidden events between the connected points. A time span can also be specified below the arrows.

VTS is intended (Alfonso et al., 2004) for — but not limited to — expressing anti-scenarios, infringements of properties.

In Fig. 15, we use time spans and the "next state" operators (< and > in the arrows) to specify anti-scenarios for p_1, p_2, p_4, p_7 and p_9.

5. Analysis

In this section, we present our analysis of the visual specification languages introduced above. We begin with a quantitative and qualitative assessment based on our running example. We then broaden

Table 8
Can the visual languages capture the properties in our running example?

	Constraint Diagram	GIL	L_R	LSC	PSC	Simulink Design Verifier	Scratch	STD[a]	TDE	Time Line Editor	VTS
G((¬Cond1 ∧ ¬Act ∧ **X** Cond1) → **X** Act)	yes		yes		yes	yes		yes			yes
`always {(Act & !Cond1)[*10]; (Reset & !Cond1)} |->{!Act}!`	yes		yes		yes	yes		yes			yes
`always {!Act;Act} |-> {Act[*10]}!`	?		yes		yes	yes		?			yes
`never {Act[*22]}`	?		yes		yes	yes		?			yes
G((¬Act ∧ **X** Act) → (¬Cond1 ∧ **X** Cond1))	yes		yes		yes	yes		yes			
G((Act ∧ Cond2) → Start)	yes	yes	yes	yes	yes	yes	yes	yes		yes	yes
`always {Start & Cond2); !Cond2[*3]} |-> {Start}!`	yes		yes		yes	yes		yes			yes
`always {(Act & !Cond2)[*4]} |-> {Stop}!`	yes		yes		yes	yes		yes			yes
G¬(Start ∧ Stop)	yes	yes	yes	yes	yes	yes	yes	yes		yes	yes
G(Start → (Act ∧ (Cond2 ∨ **Y** Cond2 ∨ **Y Y** Cond2 ∨ **Y Y Y** Cond2)))			yes			yes					
G(Stop → (¬Cond2 ∧ Act))	yes	yes	yes	yes	yes	yes	yes	yes		yes	yes
AG EF Start			yes								
AG EF ¬Start			yes								

? = We are unsure if a legal construct is possible.
[a] including STDx.

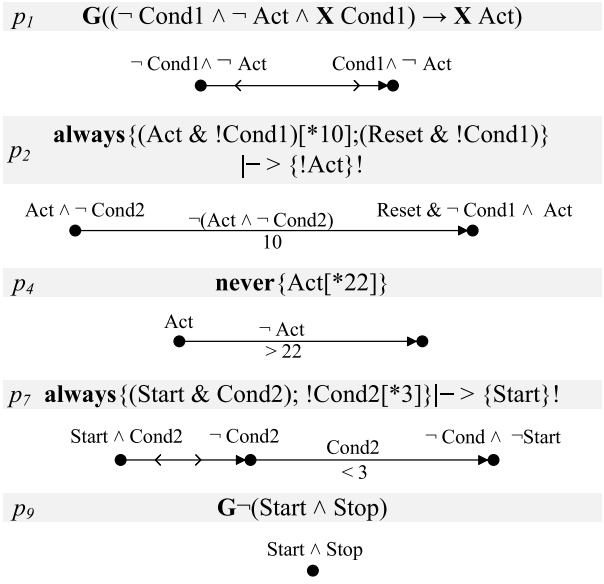

Fig. 15. Anti-scenarios for p_1, p_2, p_4, p_7 and p_9 specified using VTS.

the assessment to the properties most common in our data set. Finally, we end with a qualitative assessment based on domain specific requirements.

5.1. Evaluation based on the running example

In Table 8, we review how the different visual languages are able to capture properties in our running example from Section 3.3.

With Constraint Diagrams, we can express four of our six exemplar LTL properties, and three of our five PSL properties (see Fig. 5 for examples). However, we are unsure if our constructs for p_3 and p_4 are legal, since we use the same expression for two waveforms. The challenge is that each waveform represents a Boolean expression, and in our properties p_1, p_2, p_7, p_{10}, we cannot express the assumption waveform with just one such expression.

With GIL, we can express just p_6 and p_9, but the capabilities of the language are hardly apparent from such simple constructs (see Fig. 6). The key problem is that the **X** operator is not supported. As shown in Fig. 6, certain types of simple LTL properties with **X** operators could be expressed using a search to locate a point where a proposition changes it value, but this does not suffice for our exemplar properties.

L_R is obviously built around CTL, and the examples in Lee and Sokolsky (1997) deal with liveness properties. Also, expressing the atomic propositions of our properties as L_R "communication events" is not necessarily convenient (as is apparent from Fig. 12). In all, expressing LTL and PSL safety properties seems cumbersome.

With LSC, we run into challenges: (1) the language aims foremost at capturing liveness properties (although safety properties can be specified trough "forbidden scenarios"), and (2) there is no **X** operator or any way to otherwise express timing. In Damm and Harel (2001), the authors also state that LSC is "not intended to specify how the valuations of variables change during the runs of a system". We are therefore only able to specify our LTL properties p_6, p_9 and p_{11} with LSC, and the resulting visualizations (see. Fig. 11) are hardly useful.

With PSC, we are able to specify all of our PSL and LTL properties except p_{10}, which would have to be rewritten without the past operator **Y**.

With Simulink Design Verifier, we can specify all of our exemplar LTL and PSL properties. Liveness properties are not supported.

In Scratch, the LTL mapping suggested in Grobelna (2020) does not allow nesting of **X** or **F** statements. If **X** and **F** (along with other operators) were instead specified as "control structures" (Resnick et al., 2009) (like **G** is), the expressiveness of the language would improve, but at the cost of visual appeal. A more severe limitation is that "if then" conditional block, used for implications, only accepts Boolean logic (no temporal operators) on the "if" side, and only formulae beginning with either **X**, **F** or ¬ on the "then" side. As a result, out of our exemplar properties, we can only express p_6, p_9 and p_{11}, and p_6 and p_{11} need to be stated as "never not" or without the implication operator (e.g., "forever never (Stop and (Cond2 or not Act))" for p_{11}).

With STD, we are uncertain if constraints can be applied to a single waveform, or if the same signal can appear in different waveforms. We are therefore uncertain if our constructs for p_3 and p_4 in Fig. 9 are valid, or if the PSL properties p_2, p_7 and p_{10} can be expressed. For p_9 in Fig. 9, we have used Linear STD, a subset of STD, but in the case of simple properties like **G**p, it is difficult to see any added value. (Here, the activation condition is specified on top of the double vertical line marking the initial step.) Nevertheless, in Fig. 16, we show that **G**($q \rightarrow$ **O** q) is straightforward to specify, with a curved line specifying a precedence constraint between the q and p events.

With TDE, we cannot express any of our exemplar properties. Since the **X** operator is not supported, all of the PSL properties and three of the LTL properties cannot be specified. The LTL properties p_6, p_9, p_{11} reduce to **G**p, which we also cannot express, because the precondition

Table 9
Can the visual languages capture the most-often occurring property types from our data set?

Property type	Constraint Diagram	GIL	L_R	LSC	PSC	Simulink Design Verifier	Scratch	STD[a]	TDE	Time Line Editor	VTS
$\mathbf{G}p$	yes	yes	yes	yes	yes	yes	yes	yes		yes	yes
$\mathbf{G}(p \rightarrow \mathbf{X}\, q)$	?	?	yes		yes	yes		?			yes
`always {SERE} |-> {SERE}!`	?		yes		yes	yes		?			yes
AG EF p			yes								
`never {SERE}`	?		yes		yes	yes		?			yes
$\mathbf{G}(p \rightarrow \mathbf{O}\, q)$[b]	yes	yes	yes		yes	yes		yes		yes	yes
$\mathbf{G}\, e \rightarrow \mathbf{G}\, p$		yes	yes			yes[c]					
$\mathbf{G}(p \wedge \mathbf{X}\, q \rightarrow \mathbf{X}\, \mathbf{X}\, r)$	?		yes		yes	yes		?			yes
$\mathbf{G}(p \rightarrow \mathbf{F}\, q)$	yes	yes	yes	yes	yes		yes	yes	yes	yes	yes
$\mathbf{G}(p \wedge \mathbf{X}\, q \rightarrow r)$	?		yes		yes	yes		?			yes
$\mathbf{G}(p \rightarrow \mathbf{Y}\, q)$	?		yes		yes	yes		?			yes
`always {SERE} |=> {SERE}!`	?		yes		yes	yes		?			yes

? = Some (but not necessary all) properties of this type can be expressed.
[a] including STDx.
[b] or some equivalent expression, e.g., $\mathbf{F}p \rightarrow (\neg p \mathbf{U}(q \wedge \neg p))$ or $((\neg q)\mathbf{U}p) \vee \mathbf{G}\neg q$.
[c] $\neg e$ can be excluded through Proof Assumptions.

has to be of some length (Buzhinsky, 2018) (and we cannot require for any precondition where p is already "one" to always hold).

In TimeLine Editor, the challenges are similar to LSC. Timelines without fail events correspond to liveness properties (Smith et al., 2001). In our case study, all the LTL and PSL properties are safety properties. Out of our exemplar LTL properties, we can express p_6, p_9 and p_{11} using fail events. Since the **X** operator is not supported, the rest of our properties cannot be formulated.

With VTS, using the ability to specify the time spans, and the "next state" operator with an arrow marked with < and > in the arrow, we are able to specify anti-scenarios for all of our exemplar PSL properties, and of five our six LTL properties (all but p_5 and p_{10}).

5.2. Evaluation based on the data set

In Table 9, we review how the visual languages are able to capture the twelve most-often occurring property types in our data set, as listed in Table 4.

In some columns, we have marked "?" to signify that some — but not necessarily all — properties of the general type can be expressed. For example, as seen in Fig. 6, with GIL we can express a property like $\mathbf{G}(\neg p \wedge \mathbf{X}\, p \rightarrow \mathbf{X}\, q)$ (since we can locate the point where p changes its value), but not necessarily the more general case of $\mathbf{G}(p \wedge \mathbf{X}\, q \rightarrow \mathbf{X}\, r)$. For Constraint Diagram and STD, as we have discussed above, we are unsure if the same expression can be used on two or more waveforms.

With GIL, LSC, and TDE, the key shortcoming is the lack of a "next state" operator, or more generally, the inability to address timing. In Scratch, there is the "next" block, but the rigid nature of the language elements prevents, e.g., nesting.

The overall most difficult properties to capture are the CTL property **AG EF** p (or, "Possibility" (Campos and Machado, 2009)) and the construct $\mathbf{G}e \rightarrow \mathbf{G}(p \rightarrow q)$ (an example of the analyst wanting to exclude the states that satisfy $\neg e$).

The liveness property $\mathbf{G}(p \rightarrow \mathbf{F}q)$ can be expressed in every language save Simulink Design Verifier. Basic safety properties of the type $\mathbf{G}p$ can be expressed in every language save TDE, although LSC, PSC, Timeline Editor and VTS require the use of anti-scenarios to do so.

Specifying $\mathbf{G}(p \rightarrow \mathbf{O}\, q)$ is a good benchmark for the languages, since (1) it is the third most-popular LTL property in our data, (2) it has revealed many design issues, and (3) it highlights the convenience of the past LTL operator **O**—if p occurs, then q must have occurred "once".

In Fig. 16, we show the formulation of $\mathbf{G}(p \rightarrow \mathbf{O}\, q)$ in eight of the languages, as we are not capable of expressing the property with LSC, Scratch, or TDE. There is a case to be made for the simplicity of the LTL formulation (although we could define the "once" operator for L_R), but at least Constraint Diagram, GIL and STD provide a simple, intuitive way to state the precedence. L_R and Simulink Design Verifier expressions follow similar logic, but the solution is not necessarily intuitive. PSC, TimeLine Editor and VTS call for an anti-scenario.

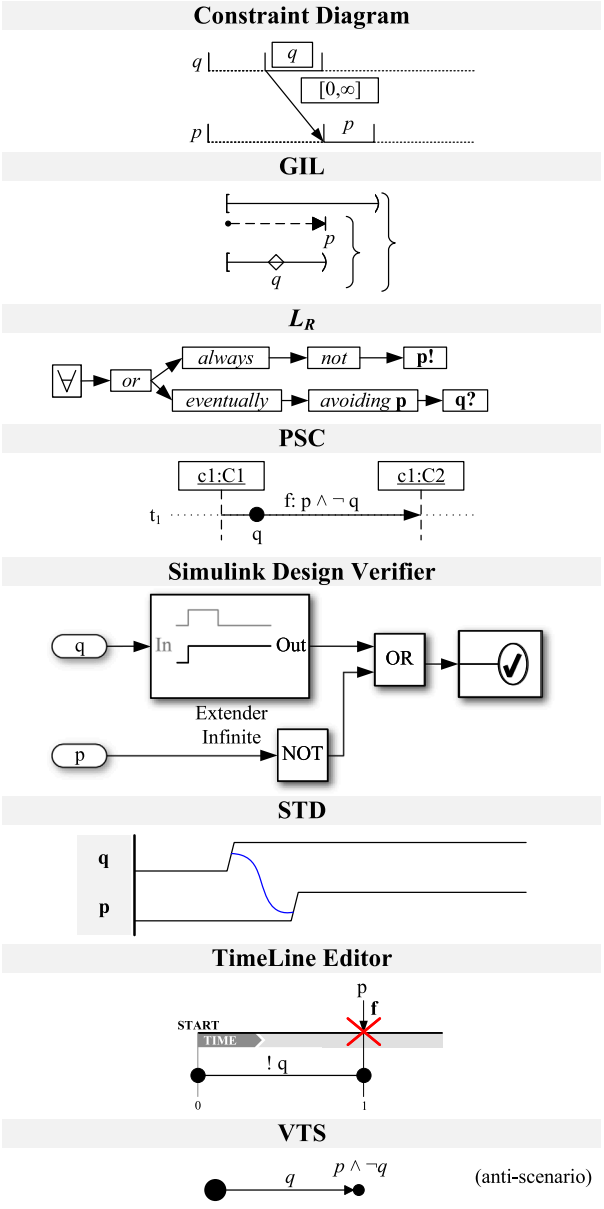

Fig. 16. $\mathbf{G}(p \rightarrow \mathbf{O}\, q)$, specified using various languages.

Table 10
Evaluation of the visual languages against desired features.

	Constraint Diagram	GIL	L_R	LSC	PSC	Simulink Design Verifier	Scratch	STD[a]	TDE	Time Line Editor	VTS
F_1 (safety and liveness)	good	good	good	fair	fair		good	good	fair	fair	good
F_2 (focus on Boolean expressions)	good	good	good			good	good	good	good		
F_3 (time in explicit terms)	good		good		good	good	good	good			good
F_4 (past time operators)	fair		fair		fair	good		good			fair
F_5 (exclusion of paths)		good	good			good					
F_6 (familiarity to I&C domain)	fair			fair	fair	good		good	good	fair	
F_7 (availability of tools)				fair	fair	good	fair		fair		

[a] including STDx.

5.3. Qualitative evaluation

Based on our practical experience, and the experiments reported in this paper, we can identify some features we would like the property specification language to have:

- F_1: The ability to express both safety and liveness properties (with emphasis on safety (Sharvia and Papadopoulos, 2015)).
- F_2: Focus on the value of Boolean expressions (rather than, e.g., messages or events).
- F_3: The ability to address time in explicit (discrete or continuous) terms.
- F_4: The ability to refer to past time.
- F_5: The easiness of excluding certain execution paths (that match a temporal property) from the analysis scope.
- F_6: Resemblance to a language already used in the industrial domain.
- F_7: Availability of practical tools that support the language.

In Table 10, we rate how well each visual language supports the above-mentioned features, using the options "good", "fair", and no support.

For F_1, Simulink Design Verifier does not support liveness properties at all, only safety. All the other languages support both. However, in LSC, PSC, TDE, and TimeLine Editor, anti-scenarios (or fail events, or fail messages) are needed to express safety properties, which can be inconvenient.

For F_2, LSC and PSC focus on messages, and TimeLine Editor and VTS on events.

For F_3, the **X** operator is not supported in GIL, LSC, TDE or TimeLine Editor. The other languages either have an explicit **X**, or some equivalent feature—"time interval" in Constraint Diagram, "next state" in GIL and VTS, time spans in VTS, the "strict operator" in PSC, the "Delay" block in Simulink Design Verifier, and the "distance measure" in STD.

For F_4, we are looking for the possibility to refer to explicit past states, so although we can express precedence and ordering in GIL and TimeLineEditor — see Fig. 16 — they do not fulfill this criterion. In both PSC and VTS, explicit references to the past are possible, although a **Y** operator as such is missing. In L_R, the user is free to specify keywords for past operators. Simulink Design Verifier has a **Y** operator in the form of the "Delay" block, and STD has the "precedence" and "distance measure" features.

For F_5, Simulink Design Verifies provides "Proof Assumptions" as a way to exclude certain execution paths from the analysis scope. In GIL, we can specify an interval that excludes execution paths. For the other languages, we did not identify similarly convenient mechanisms.

For F_6, the function blocks used in Simulink Design Verifier are clearly familiar to I&C engineers. We also find the kind of waveforms employed in STD and TDE common in the visualization of, e.g., trend graphs (or, in our own case, counterexamples in Section 3.3). The UML style sequence diagrams of LSC and PSC, and the timing diagrams of Constraint Diagram and TimeLine Editor also at least bear resemblance to modeling languages used in the domain.

Finally, for F_7, tool support seems scarce, with the exception of Simulink Design Verifier, which is a commercial product available from MathWorks. For Constraint Diagram, a 2002 paper (Dierks and Lettrari, 2002) mentions a prototype tool called Moby/CD, which does not seem to be available. As we mention in Section 2.4, A tool from 2016 called SESAMM Specifier (Filipovikj et al., 2016) supports RTGIL and L_R, but that tool is not publicly available, either. For LSC, a programming tool called PlayGo is freely available (Weizmann Institute of Science, 2020), but does not seem to support translation of properties to, e.g., LTL. PragmaDev Tracer (PragmaDev, 2023) is a freely available tool for specifying properties in PSC, but the use of the properties seems to be limited to checking traces of systems modeled in PragmaDev's own environment. Scratch is a common-purpose language, with plenty of free tools available, but no tools support the property specification features suggested in Grobelna (2020). Two tools, (1) a STD tool called CheckOff (Schlör, 2002), and (2) a tool for TimeLine Editor (from Bell Labs) (Smith, 2004) have at some point been available, but are no longer found online. A tool supporting TDE is still available (Vyatkin, 2007), but the implementation is from 2007, and left to a prototype stage. For VTS, support is limited to MS Visio templates (Univesity of Buenos Aires, 2006).

6. Discussion

6.1. Applicability of visual languages

Based on our quantitative analysis against our property sets, L_R appears as the most capable language. However, the downside of the expressive power is that the language mostly serves as syntactic sugar (for CTL). It is hard to claim that our exemplar graphs in Figs. 12 and 16 would be easier to grasp than the original LTL and PSL formulae.

PSC, Simulink Design Verifier and VTS can also capture the most of our properties, but in our qualitative analysis, PSC and VTS fall short.

PSC and VTS are both scenario-based languages, PSC depicting messages sent between components, and VTS focusing on events in a more general sense. As we are interested in the values that Boolean expressions on variables have over time, it is cumbersome to specify the properties in terms of PSC messages or VTS events. Instead of (wanted or unwanted) scenarios, verifiable properties should focus on transitions between system states in reactive system not designed to terminate (Clarke et al., 1999).

Furthermore, to express safety properties in PSC or VTS, the analyst needs to specify anti-scenarios. Whereas some of our exemplar properties readily describe the anti-scenario (e.g., $\mathbf{G}(\neg p)$), more generally, having to translate each safety property to its anti-scenario(s) leaves room for errors and omissions. The languages also lack an easy way to exclude certain scenarios from analysis, which — as we discussed in Section 3.2 — is a often useful in, e.g., filtering irrelevant counterexamples.

Combining all of our results, it would seem that Simulink Design Verifier holds the most potential. The language seems very appealing, given that it is based on function block diagrams. "Proof Assumptions" provide an easy way to exclude scenarios from analysis. Simulink Design Verifier is an established and a commercially available tool.

Another benefit of the Design Verifier tool is that it also animates the property when showing a counterexample. Such a feature helps the user to better understand the property, and realize potential errors made in its specification. We explore similar animation of LTL properties in Pakonen et al. (2021b).

The concerns we have are:

1. The analyst needs to learn the logic of the temporal operator blocks.
2. The design and its specification are expressed in a very similar manner, which might lead to confusion.
3. Tool support is limited to the Simulink product family, and modeling nuclear, vendor-specific, non-standard I&C block diagrams with Simulink with 100% accuracy might not be straightforward. (Consider, for example, the notion of signal validity (Pakonen et al., 2021a; Pakonen and Björkman, 2017).)
4. Liveness properties are not supported.

It is also debatable (and certainly a matter of personal preference) if the diagrams we show in Figs. 14 and 16 are actually easier to write and read than the corresponding LTL/PSL formulae.

In our qualitative feature analysis, STD and Constraint Diagram got reasonable results. Both are based on timing diagrams, but provide a way to specify explicit timing constraints (at least in the STDx variant), and allow references to past events. However, it is far from clear how many of our PSL properties we could actually state with either language, as we cannot necessarily express all the different propositions that appear in every SERE as constraints between individual waveforms. For example, we are unsure if our expressions for p_3 and p_4 in Figs. 5 and 9 are actually allowed.

Another limitation with STD, in particular, is that the activation condition needs to match the value of every waveform. For example, considering the property p_8 in Fig. 9: the activation condition specifies that the signal "Stop" is false. If there is a relevant counterexample for our original requirement where "Stop" is first true, the scenario will not then be matched. The "any" value available in TDE for the waveform signals would remove this limitation.

GIL, LSC, TDE, and TimeLine Editor lack a "next" operator, or other mechanism to refer explicitly to a specific past or future state. Scratch has a "next" element, but nesting the elements is not possible. This is a severe limitation, and it means that the languages are not able to express many our property types.

GIL also does not resemble any language that would already be familiar to I&C engineers. And although GIL (or RTGIL) has been used in some proposed tools (e.g., Filipovikj et al. (2016)), no such tool seems to be available.

LSC (based on sequence charts) and TimeLine Editor (based on timing diagrams) are aimed at expressing liveness properties, and anti-scenarios are again the only option to express the safety properties that are more common for us.

Scratch provides a different and arguably entry-level approach to property specification, but the blocks suggested in Grobelna (2020) are not at all sufficiently flexible for our use. Still, a similar approach with differently specified building blocks might be an idea worth refining.

Finally, with TDE, we were only able to capture the liveness property $\mathbf{G}(p \rightarrow \mathbf{F}\, q)$. Still, as stated above, the ability to assign "any" (or "stable") as the signal value on the waveform is an idea we would also like to see implemented in, e.g., STD.

6.2. On specification patterns

As stated above, specification patterns are commonly suggested solution to make property specification user-friendly. Our data set — as can be seen in Table 4 — also has fairly small number of oft-occurring property types, which we could consider candidates for such patterns. In the following, we therefore examine how well the properties in our data set could be captured using different suggested property pattern collections.

In Table 11, we show the occurrence of the most common LTL and CTL property types in our data set in selected pattern collections. Compared to Table 4, we distinguish between $\mathbf{G}p$ and $\mathbf{G}(\neg p)$ since, although the temporal structure is the same, many collections have a separate pattern for $\mathbf{G}(\neg p)$ (called, e.g. "Absence" (Dwyer et al., 1999)). In Table 12, we further break down the variants of $\mathbf{G}p$ in our data, to better compare against the patterns in Menghi et al. (2021).

Dwyer et al. claim that in their experience, 92% of collected requirements match their specification patterns, the most popular being Response ("p leads to q"), Universality ("p is true"), and Absence ("p is false") (Dwyer et al., 1999). When we compare to our data, we can see that 55% of our properties fall under Universality, Absence, Precedence ("q enables p"), and Response. (If we only count our LTL and CTL specifications, the share is 65%.)

We get better coverage with the I&C specific patterns of Campos et al. (2008), since the patterns include Possibility ($\mathbf{AG\ EF}\ p$) and Immediate Response ($\mathbf{G}(p \rightarrow \mathbf{X}\, q)$), our second and third most common types, bringing the total up to 76% (or 89% if we only count LTL and CTL).

The pattern $\mathbf{G}(p \rightarrow \mathbf{X}\, q)$ is included in every collection except Dwyer et al. and Monteiro et al. (2008).

In Table 11, we also list the closed-loop patterns identified in Buzhinsky (2018). We see that, e.g., the "Infinitely often" pattern ($\mathbf{G\,F}\, p$) also appears in Campos et al. (2008), Menghi et al. (2021) and Zhang et al. (2021). To clarify why such properties do not appear in our data, let us consider a requirement that a nuclear reactor shall "infinitely often" be in a state where the pressure is within normal limits. In the closed-loop model, this property is relevant, as it ensures that the I&C is always capable of bringing the plant to a safe state. However, in the open-loop model, the I&C system cannot be specified to "infinitely often" react to anything, because the counterexample would be an infinite loop where the pressure is never high. This fact might also explain why, for example, the operator $\mathbf{U}$ is relatively rarely used in our data, since $p\,\mathbf{U}\,q$ is false if q never occurs.

Although both analysts A and B behind our data have been aware of the patterns of Dwyer et al. for most of their work history, they made no concentrated effort to systematically apply them in their work.

If we were to introduce a pattern collection of our own, based on our data, we can count from Table 4 that ten patterns could capture 94% of all our collected LTL and CTL properties (or 80% of all properties). But after the six most common patterns, any new pattern would only increase the coverage by less than 0.1%. And, in our data, there are 75 LTL/CTL property types that appear only once.

If we wish to capture more, in a way that still frees the user from having to learn temporal logic, the number of the necessary patterns increases dramatically. Looking at our data, the increase would be due to (1) timing aspects we currently capture with PSL SEREs, and (2) variations seen especially in Table 5 where we have highlighted the "$\mathbf{G}\,e \rightarrow$" and "$i \rightarrow$" additions we have used to exclude certain execution paths.

In the patterns of Dwyer et al. "scopes" are used to determine a starting and ending event for the pattern (Dwyer et al., 1998). The focus is more on ordering of states rather their explicit timing, but a similar mechanism could be used to include the "$\mathbf{G}\,e \rightarrow$" and "$i \rightarrow$" variations.

Nevertheless, the notable share of the PSL properties in our data (14.3%) suggests that a pattern-based solution alone (without a SERE-type logic for expressing sequences of states, at least) does not suffice for our domain. In general, it is also challenging to create specification patterns that are both generic enough to be applicable, and concrete enough to be comprehensible (Monteiro et al., 2008).

Table 11
Occurrence of our LTL and CTL property types in proposed pattern collections.

Dwyer et al. (1999)	Campos et al. (2008)	Menghi et al. (2021)	Zhang et al. (2021)	Monteiro et al. (2008)	Closed-loop Buzhinsky (2018)	Mapping	Freq. %
Universality	Universality	(Table 12)	Universality	Invariance	Always	$\mathbf{G}\ p$	48.0[a]
	Immediate response	Prompt reaction	Immediately follow		Response on the next step	$\mathbf{G}(p \rightarrow \mathbf{X}\ q)$[d]	11.0
	Possibility					$\mathbf{AG}\,\mathbf{EF}\ p$	9.5
Absence	Absence	Global avoidance	Absence	(Invariance)	(Always)	$\mathbf{G}\ \neg p$	4.0
Precedence	Precedence				Cannot happen until	$\mathbf{G}(p \rightarrow \mathbf{O}\ q)$[b]	2.7
						$\mathbf{G}\ e \rightarrow \mathbf{G}\ p$	1.7
						$\mathbf{G}(p \wedge \mathbf{X}\ q \rightarrow \mathbf{X}\,\mathbf{X}\ r)$	1.0
Response	Eventual response	Delayed reaction		Consequence	Unbounded response	$\mathbf{G}(p \rightarrow \mathbf{F}\ q)$[c]	0.8
						$\mathbf{G}(p \wedge \mathbf{X}\ q \rightarrow r)$	0.6
						$\mathbf{G}(p \rightarrow \mathbf{Y}\ q)$	0.5
	Fairness	Patrolling	Frequency		Infinitely often	$\mathbf{G}\,\mathbf{F}\ p$	-
					Infinitely often after condition	$\mathbf{G}(p \rightarrow \mathbf{G}\,\mathbf{F}\ q)$	-
				Occurrence	Reachable	$\mathbf{EF}\ p$	-[e]
Existence		Visit	Existence		Eventually	$\mathbf{F}\ p$	-

[a] Does not include $\mathbf{G}(\neg p)$.
[b] In Dwyer et al. (1999) and Campos et al. (2008), $\mathbf{F}\ p \rightarrow (\neg p\ \mathbf{U}\ (q \wedge \neg p))$. In Buzhinsky (2018), $((\neg q)\ \mathbf{U}\ p) \vee \mathbf{G}\ \neg q$.
[c] In Monteiro et al. (2008), $\mathbf{AG}(p \rightarrow \mathbf{AF}\ q)$.
[d] In Campos et al. (2008), $\mathbf{AG}(p \rightarrow \mathbf{AX}\ q)$.
[e] Occurs once.

Table 12
Temporally equivalent variants of $\mathbf{G}(p)$ appearing in our data.

Variant	Menghi et al. (2021)	Count	%
$\mathbf{G}(a \rightarrow b)$	Inst. Reaction	1724	44
$\mathbf{G}\ \neg a$	Global Avoidance	157	4.0
$\mathbf{G}(a \leftrightarrow b)$	Bound Reaction	82	2.1
$\mathbf{G}\ a$		70	1.8

6.3. On textual languages

PSL, although a textual language, is intended as user-friendly. Looking at our sample data, could the analysts have done even more of the specifications with PSL?

Most LTL property types listed in Table 4 could be expressed using PSL. The lack of past temporal operators (beyond `before` and its variants), however, means that many properties would have to be reformulated. With `before_`, we can, for example, specify type (6) in Table 4, $\mathbf{G}(p \rightarrow \mathbf{O}\ q)$, as: `always (q before_ p)`.

In Ljungkrantz et al. (2014), the authors propose a property specification language called ST-LTL, based on the IEC 61131-3 programming language Structured Text (ST). ST-LTL is in effect syntactic sugar for LTL:

- Main operators such as `ALWAYS`, `EVENTUALLY` and `implies` replace **G**, **F**, and →.
- Suffixes `p_previous`, `p_risingEdge` and `p_fallingEdge` replace **Y** p, $(\neg p \wedge \mathbf{X}\ p)$ and $(p \wedge \mathbf{X}\ \neg p)$.
- Functions such as `DuringNScanCycles(p,n)` (stating that `p` shall be true for `n` cycles) help specify timed behavior.

With ST-LTL, we can capture some of the properties from our running example (e.g., $p4$: `NEVER (DuringNScanCycles(Act,22))`, or $p5$:
`ALWAYS (Act_risingEdge IMPLIES Cond1_risingEdge))`, but the language is not as flexible or (in our opinion) easy to read as PSL.

6.4. Validity of results

Regarding the validity of our experiments, a major concern is that we cannot be sure of the correctness of the properties as we have expressed them using the different visual languages, which threatens the validity of our quantitative claims, in particular. We did not have practical tools for the languages on disposal. Such tools would have revealed syntactical errors, and actual verification with a model checker could also have revealed logical errors in our constructs. This limitation is clear in the cases where — based on the limited information we could find — we admit to be unsure if a legal construct is possible.

Another key concern is that the dataset we collected comes from the work of just two analysts, and Analyst A is responsible for 92% of the written properties. We can identify different traits between the analysts (partially attributable to their respective backgrounds)—A has a tendency to use PSL more often, whereas B is more comfortable with complex LTL. Still, the patterns most often used by Analyst B are also common for Analyst A, although the personal preferences differ (see Table 13.)

Still, there remains a concern over how generally representative our dataset really is. If Analyst A, for example, had a university degree in formal methods, the data could appear very different.

When we compare our data against the closed-loop patterns listed in Buzhinsky (2018), the differences may not all be due to the distinction between open and closed-loop modeling. (The closed-loop properties were written by a researcher not involved in VTT's practical industry projects.) Still, we can at least claim that the practical impact of the work that analysts A and B have done — 109 real-world design issues revealed in successful practical projects, in two quite different industrial domains — means that the properties we have collected have proven useful in an actual industrial setting.

Still regarding the data, not all of the collected properties necessarily relate to actual functional system requirements, but may represent an attempt by the analyst to understand some aspect of the system (model) behavior. The analyst might even have written a property they expect to fail, in order to ensure their understanding is correct. In any case, all the properties have been deemed useful by the respective analyst in their work, so therefore we can argue that having a user-friendly way to specify such properties would be of use.

NuSMV only supports the verification of PSL specifications for a subset of PSL, and generally, the limitations of NuSMV might have

Table 13
The most common types of properties written by Analyst B, and their occurrence in the work of Analyst A.

#	Generalized type	Analyst B	Analyst A
1	$\mathbf{G}\ p$	67%	50%
2	$\mathbf{G}(p \rightarrow \mathbf{X}\ q)$	8.2%	11.6%
3	$\mathbf{G}(p \rightarrow \mathbf{F}\ q)$	6.1%	0.3%
4	`always {`*SERE*`} |-> {`*SERE*`}!`	4.0%	11.2%
5	$\mathbf{G}(p \wedge \mathbf{X}\ q \rightarrow \mathbf{X}\,\mathbf{X}\ r)$	1.5%	1.0%

introduced restrictions to property types appearing in our data that CTL, LTL and PSL do not otherwise impose.

When assessing the applicability of a given language for a task (here, formal property specification), the optimal approach would be to learn the language, then try to apply that language to the task at hand. Here, we instead collected statements already expressed in some formal language (LTL, CTL and PSL), and tried translating those statements to the languages we reviewed. Free experimentation, starting from the original, natural language functional requirements, was not possible.

7. Conclusions

Temporal property specification is hard, and making it easier through different approaches has been a research topic for decades.

The "classic" solution is to apply specification patterns. The sample data we have collected from VTT's practical model-checking projects support the conclusions others have already drawn: a relatively small number of patterns can cover a large part of the specification needs. On the other hand, the data support our own assumption that a specification language aimed the verification of I&C logics needs also to support properties that deal with timing issues and sequences. 32% of all sampled properties referred to certain specific (next or previous) states, and we cannot assign every variation its own pattern.

With visual specification languages, there is always a trade-off between expressiveness and accessibility. A very expressive language like L_R does not necessarily free the user from having to study the fundamentals of CTL serving as its basis. Conversely, the more a language resembles timing or sequence diagrams — familiar to I&C professionals — the less power it has in capturing complex temporal properties.

The silver bullet remains elusive. Of the eleven visual languages we experimented, not one was perfect for our purposes. Many of the languages were relatively old by now, some having already been suggested in the 1990s. In many approaches, the focus is on liveness properties, or messages sent between components, rather than on transitions between states in a reactive system, which is where we would like it to be. (And where it arguably (Clarke et al., 1999) needs to be.)

Still, even the language we found least capable in expressing our properties (TDE) contained interesting and usable ideas. And overall, the function block approach of Simulink Design Verifier strikes a nice balance between being familiar to an I&C expert, and capable of expressing most types of properties relevant to us.

We can capture 77% of our collected properties with just six LTL/CTL temporal structures: (1) $\mathbf{G}\ p$, (2) $\mathbf{G}(p \rightarrow \mathbf{X}\ q)$ (3) $\mathbf{AG}\ \mathbf{EF}\ p$, (4) $\mathbf{G}(p \rightarrow \mathbf{O}\ q)$, (5) $\mathbf{G}\ e \rightarrow \mathbf{G}\ p$, and (6) $\mathbf{G}(p \wedge \mathbf{X}\ q \rightarrow \mathbf{X}\,\mathbf{X}\ r)$. After that, any new pattern would add less than 1% to the overall coverage. PSL properties cover an additional 14%. These numbers imply that a quite limited collection of patterns, along with SERE-type PSL, would already be a useful combination. Also, a visual language (and a tool) based on easy manipulation of SERE-type properties would seem like a worthwhile topic for further research.

On the other hand, in Section 3.2, we listed the property types that have actually revealed design issues, and the list includes many types that occur only once in our data (so they would not qualify as pattern candidates), and contain fairly complex temporal formulae, also including past temporal operators like $\mathbf{Y}$ not supported by PSL.

We hope that the properties we have collected — and the desired language features we list — prove useful in the development of different approaches on user-friendly property specification. After decades of research, had the problem been already solved, formal methods such as model checking could today be in much wider use.

CRediT authorship contribution statement

Antti Pakonen: Conceptualization, Data curation, Funding acquisition, Investigation, Methodology, Project administration, Visualization, Writing – original draft, Writing – review & editing. **Igor Buzhinsky:** Conceptualization, Data curation, Investigation, Methodology, Writing – review & editing. **Valeriy Vyatkin:** Conceptualization, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Formal properties collected from practical model-checking projects (Original data) (Zenodo).

References

Adiego, B.F., Darvas, D., Viñuela, E.B., Tournier, J.-C., Bliudze, S., Blech, J.O., Suárez, V.M.G., 2015. Applying model checking to industrial-sized PLC programs. IEEE Trans. Ind. Inform. 11 (6), 1400–1410. http://dx.doi.org/10.1109/TII.2015.2489184.

Alfonso, A., Braberman, V., Kicillof, N., Olivero, A., 2004. Visual timed event scenarios. In: Proc. ICSE. Edinburgh, UK, pp. 168–177. http://dx.doi.org/10.1109/ICSE.2004.1317439.

Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A., 2015. Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar. IEEE Trans. Softw. Eng. 41 (7), 620–638. http://dx.doi.org/10.1109/TSE.2015.2398877.

Autili, M., Inverardi, P., Pelliccione, P., 2007. Graphical scenarios for specifying temporal properties: an automated approach. Autom. Softw. Eng. 14, 293–340. http://dx.doi.org/10.1007/s10515-007-0012-6.

Behrmann, G., David, A., Larsen, K.G., 2004. A tutorial on Uppaal. In: Formal Methods for the Design of Real-Time Systems (SFM-RT 2004). In: LNCS, (no. 3185), Springer Berlin Heidelberg, pp. 200–236. http://dx.doi.org/10.1007/978-3-540-30080-9_7.

Bel V, BASE, CNSC, CSN, ISTec, KAERI, KINS, NSC, ONR, SSM, STUK, 2018. Licensing of safety critical software for nuclear regulators, Common position of international nuclear regulators and authorised technical support organisation. Common position Revision 2018, URL http://www.onr.org.uk/software.pdf.

Benedetti, M., Cimatti, A., 2003. Bounded model checking for past LTL. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003). In: LNCS, (no. 2619), Springer Berlin Heidelberg, pp. 18–33. http://dx.doi.org/10.1007/3-540-36577-X_3.

Boyer, P., 2009. Model-checking timed temporal logics. Electron. Notes Theor. Comput. Sci. 231, 323–341. http://dx.doi.org/10.1016/j.entcs.2009.02.044.

Bozzano, M., Cimatti, A., Katoen, J.-P., Katsaros, P., Mokos, K., Nguyen, V., Noll, T., Postma, B., Roveri, M., 2014. Spacecraft early design validation using formal methods. Reliab. Eng. Syst. 132, 20–35. http://dx.doi.org/10.1016/j.ress.2014.07.003.

Brunello, A., Montanari, A., Reynolds, M., 2019. Synthesis of LTL formulas from natural language texts: state of the art and research directions. In: Proc. TIME. Málaga, Spain, pp. 17:1–17:19. http://dx.doi.org/10.4230/LIPIcs.TIME.2019.17.

Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J., 1992. Symbolic model checking: 10^{20} states and beyond. Inform. and Comput. 98 (2), 142–170. http://dx.doi.org/10.1016/0890-5401(92)90017-A.

Buzhinsky, I., 2018. Visual formalisms applied to temporal logic property patterns in the nuclear automation domain. Research report, Aalto University, http://dx.doi.org/10.13140/RG.2.2.26467.20009.

Buzhinsky, I., Pakonen, A., 2020. Timed model checking of fault-tolerant nuclear I&C systems. In: Proc. INDIN. Warwick, UK, pp. 159–164. http://dx.doi.org/10.1109/INDIN45582.2020.9442188.

Campos, J.C., Machado, J., 2009. Pattern-based analysis of automated production systems. IFAC Proc. Vol. 42 (4), 972–977. http://dx.doi.org/10.3182/20090603-3-RU-2001.0425.

Campos, J.C., Machado, J., Seabra, E., 2008. Property patterns for the formal verification of automated production systems. IFAC Proc. Vol. 41 (2), 5107–5112. http://dx.doi.org/10.3182/20080706-5-KR-1001.00858.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A., 2002. NuSMV 2: An open source tool for symbolic model checking. In: Computer Aided Verification (CAV 2002). In: LNCS, (no. 2404), Springer Berlin Heidelberg, pp. 359–364. http://dx.doi.org/10.1007/3-540-45657-0_29.

Cimatti, A., Griggio, A., Mover, S., Tonetta, S., 2004. HyComp: An SMT-based model checker for hybrid systems. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015). In: LNCS, (no. 9035), Springer Berlin Heidelberg, pp. 52–67. http://dx.doi.org/10.1007/978-3-662-46681-0_4.

Clarke, E., Biere, A., Raimi, R., Zhu, Y., 2001. Bounded model checking using satisfiability solving. Form. Methods Syst. Des. 19, 7–34. http://dx.doi.org/10.1023/A:1011276507260.

Clarke, E., Grumber, O., Peled, D., 1999. Model checking. MIT Press, Cambridge, Massachusetts, US.

Czepa, C., Zdun, U., 2020. On the understandability of temporal properties formalized in linear temporal logic, property specification patterns and event processing language. IEEE Trans. Softw. Eng. 46 (1), 100–112. http://dx.doi.org/10.1109/TSE.2018.2859926.

Damm, W., Harel, D., 2001. LSCs: Breathing life into message sequence charts. Form. Methods Syst. Des. 19, 45–80. http://dx.doi.org/10.1023/A:1011227529550.

Dierks, H., Lettrari, M., 2002. Constructing test automata from graphical real-time requirements. In: Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2002). In: LNCS, (no. 2469), Springer Berlin Heidelberg, pp. 433–453. http://dx.doi.org/10.1007/3-540-45739-9_25.

Dietz, C., 1996. Graphical formalization of real-time requirements. In: Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 1996). In: LNCS, (no. 1135), Springer Berlin Heidelberg, pp. 366–384. http://dx.doi.org/10.1007/3-540-61648-9_51.

Dillon, L.K., Kutty, G., Moser, L.E., Melliar-Smith, P.M., Ramakrishna, Y.S., 1994. A graphical interval logic for specifying concurrent systems. ACM Trans. Softw. Eng. Methodol. 3 (2), 131–165. http://dx.doi.org/10.1145/192218.192226.

Dwyer, M.B., Avrunin, G.S., Corbett, J.C., 1998. Property specification patterns for finite-state verification. In: Proc. FMSP. New York, NY, USA, pp. 7–15. http://dx.doi.org/10.1145/298595.298598.

Dwyer, M.B., Avrunin, G.S., Corbett, J.C., 1999. Patterns in property specifications for finite-state verification. In: Proc. ICSE. New York, NY, USA, pp. 411–420. http://dx.doi.org/10.1145/302405.302672.

Eisner, C., Fisman, D., 2006. A Practical Introduction to PSL. Springer, US.

Filipovikj, P., Jagerfield, T., Nyberg, M., Rodriguez-Navas, G., Seceleanu, C., 2016. Integrating pattern-based formal requirements specification in an industrial tool-chain. In: Proc. COMPSAC. Atlanta, GA, USA, pp. 167–173. http://dx.doi.org/10.1109/COMPSAC.2016.140.

Fockel, M., Holtmann, J., Koch, T., Schmelter, D., 2018. Formal, model- and scenario-based requirement patterns. In: Proc. MODELSWARD. Funchal, Portugal, pp. 331–318. http://dx.doi.org/10.5220/0006554103110318.

Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J., 2020. Generation of formal requirements from structured natural language. In: Requirements Engineering: Foundation for Software Quality (REFSQ 2020). In: LNCS, (no. 12045), Springer Cham, pp. 19–35. http://dx.doi.org/10.1007/978-3-030-44429-7_2.

Grobelna, I., 2020. Scratch-based user-friendly requirements definition for formal verification of control systems. Inform. Educ. 19 (2), 223–238. http://dx.doi.org/10.15388/infedu.2020.11.

Guo, X., Aoki, T., Lin, H.-H., 2020. Model checking of in-vehicle networking systems with CAN and FlexRay. J. Syst. Softw. 161, 110461. http://dx.doi.org/10.1016/j.jss.2019.110461.

Holzmann, G.J., 2002. The logic of bugs. In: Proc. SIGSOFT/FSE. Charleston, SC, USA, pp. 81–87. http://dx.doi.org/10.1145/587051.587064.

IAEA, 2021. Introduction to systems engineering for the instrumentation and control of nuclear facilities. Nuclear Energy Series NP-T-2.14, International Atomic Energy Agency, URL https://www-pub.iaea.org/MTCD/Publications/PDF/PUB2018_web.pdf.

IEC, 2012. Property Specification Language (PSL). IEC Standard 62531:2012, International Electrotechnical Commission.

IEC, 2013. Programmable Controllers, Part 3: Programming Languages. IEC Standard 61131-3:2013, International Electrotechnical Commission.

Konrad, S., Cheng, B.H.C., 2005. Real-time specification patterns. In: Proc. ICSE. St. Louis, MO, USA, pp. 372–381. http://dx.doi.org/10.1145/1062455.1062526.

Lamport, L., 1977. Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. SE-3 (2), 125–143. http://dx.doi.org/10.1109/TSE.1977.229904.

Lee, I., Sokolsky, O., 1997. A graphical property specification language. In: Proc. HASE. Washington, DC, USA, pp. 42–47. http://dx.doi.org/10.1109/HASE.1997.648037.

Ljungkrantz, O., Åkesson, K., Fabian, M., Ebrahimi, A.H., 2014. An empirical study of control logic specifications for programmable logic controllers. Empir. Softw. Eng. 19, 655–677. http://dx.doi.org/10.1007/s10664-012-9232-x.

Ljungkrantz, O., Åkesson, K., Fabian, M., Yuan, C., 2010. A formal specification language for PLC-based control logic. In: Proc. INDIN. Osaka, Japan, pp. 1067–1072. http://dx.doi.org/10.1109/INDIN.2010.5549591.

Mathworks, 2023. Simulink design verifier. URL https://www.mathworks.com/products/simulink-design-verifier.html.

Mavin, A., Wilkinson, P., Harwood, A., Novak, M., 2009. Easy approach to requirements syntax (EARS). In: Proc. RE. Atlanta, GA, USA, pp. 317–322. http://dx.doi.org/10.1109/RE.2009.9.

Menghi, C., Tsigkanos, C., Pelliccione, P., Ghezzi, C., Berger, T., 2021. Specification patterns for robotic missions. IEEE Trans. Softw. Eng. 47 (10), 2208–2224. http://dx.doi.org/10.1109/TSE.2019.2945329.

Menshawy, M.E., Bentahar, J., Kholy, W.E., Laarej, A., 2018. Model checking real-time conditional commitment logic using transformation. J. Syst. Softw. 138, 189–205. http://dx.doi.org/10.1016/j.jss.2017.12.042.

Monteiro, P.T., Ropers, D., Mateescu, R., Freitas, A.T., de Jong, H., 2008. Temporal logic patterns for querying dynamic models of cellular interaction networks. Bioinformatics 24 (16), 227–233. http://dx.doi.org/10.1093/bioinformatics/btn275.

Németh, E., Bartha, T., 2008. Formal verification of safety functions by reinterpretation of functional block based specifications. In: Formal Methods for Industrial Critical Systems (FMICS 2008). In: LNCS, (no. 5596), Springer Berlin Heidelberg, pp. 199–214. http://dx.doi.org/10.1007/978-3-642-03240-0_17.

Nikora, A.P., Balcom, G., 2009. Automated identification of LTL patterns in natural language requirements. In: Proc. ISSRE. Mysuru, India, pp. 185–194. http://dx.doi.org/10.1109/ISSRE.2009.15.

Pakonen, A., 2021a. Model-checking of I&C logics—insights from over a decade of projects in Finland. In: Proc. NPIC&HMIT. Providence, RI, USA, pp. 792–801. http://dx.doi.org/10.13182/T124-34322.

Pakonen, A., 2021b. Oops! Examples of I&C design issues detected with model checking. In: Proc. ISOFIC. Okoyama, Japan, URL https://cris.vtt.fi/files/53941549/Pakonen_ISOFIC_2021_.pdf.

Pakonen, A., 2023. Formal properties collected from practical model-checking projects, v1. http://dx.doi.org/10.5281/zenodo.7759742.

Pakonen, A., Björkman, K., 2017. Model checking as a protective method against spurious actuation of industrial control systems. In: Safety and Reliability. Theory and Applications. CRC Press, pp. 3189–3196.

Pakonen, A., Buzhinsky, I., Björkman, K., 2021a. Model checking reveals design issues leading to spurious actuation of nuclear instrumentation and control systems. Reliab. Eng. Syst. 205, 107237. http://dx.doi.org/10.1016/j.ress.2020.107237.

Pakonen, A., Buzhinsky, I., Vyatkin, V., 2021b. Counterexample visualization and explanation for function block diagrams. In: Proc. INDIN. Porto, Portugal, pp. 747–753. http://dx.doi.org/10.1109/INDIN.2018.8472025.

Pakonen, A., Pang, C., Buzhinsky, I., Vyatkin, V., 2016. User-friendly formal specification languages – conclusions drawn from industrial experience on model checking. In: Proc. ETFA. Berlin, Germany, http://dx.doi.org/10.1109/ETFA.2016.7733717.

Pavlovic, O., Ehrich, H.-D., 2010. Model checking PLC software written in function block diagram. In: Proc. ICST. Paris, France, pp. 439–448. http://dx.doi.org/10.1109/ICST.2010.10.

PragmaDev, 2023. PragmaDev tracer. URL https://www.pragmadev.com/product/tracer.html.

Preuße, S., Lapp, H., Hanisch, H., 2012. Closed-loop system modeling, validation, and verification. In: Proc. ETFA. Krakow, Poland, http://dx.doi.org/10.1109/ETFA.2012.6489679.

Ramakrishna, Y.S., Melliar-Smith, P.M., Moser, L.E., Dillon, L.K., Kutty, G., 1996. Interval logics and their decision procedures: Part II: a real-time interval logic. Theoret. Comput. Sci. 170 (1–2), 1–46. http://dx.doi.org/10.1016/S0304-3975(96)80701-8.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y., 2009. Scratch: Programming for all. Commun. ACM 52 (11), 60–67. http://dx.doi.org/10.1145/1592761.1592779.

Santos, T., Carvalho, G., Sampaio, A., 2018. Formal modelling of environment restrictions from natural-language requirements. In: Formal Methods: Foundations and Applications (SBMF 2018). In: LNCS, (no. 11254), Springer Cham, pp. 252–270. http://dx.doi.org/10.1007/978-3-030-03044-5_16.

Schlör, R., 2002. Symbolic timing diagrams: a visual formalism for model verification (Ph.D. thesis). C. v. O. Univ., Oldenburg, Germany.

Schlör, R., Josko, B., Werth, D., 1998. Using a visual formalism for design verification in industrial environments. In: Services and Visualization Towards User-Friendly Design. In: LNCS, (no. 1385), Springer Berlin Heidelberg, pp. 208–221. http://dx.doi.org/10.1007/BFb0053507.

Sharvia, S., Papadopoulos, Y., 2015. Integrating model checking with HiP-HOPS in model-based safety analysis. Reliab. Eng. Syst. 135, 64–80. http://dx.doi.org/10.1016/j.ress.2014.10.025.

SLAC National Accelerator Laboratory, 2007. What's new in release 2007b. URL https://www.slac.stanford.edu/grp/md/ecae/Release_Notes_Matlab_R2007b.html.

Smith, M., 2004. Timeedit, a visual tool for specifying correctness properties. URL http://www.spinroot.com/spin/Doc/course/timelineeditor.pdf.

Smith, M., Holzmann, G., Etessami, K., 2001. Events and constraints: a graphical editor for capturing logic requirements of programs. In: Proc. RE. Toronto, ON, Canada, pp. 14–22. http://dx.doi.org/10.1109/ISRE.2001.948539.

Stewart, D., Liu, J., Cofer, D., Heimdahl, M., Whalen, M., Peterson, M., 2021. AADL-based safety analysis using formal methods applied to aircraft digital systems. Reliab. Eng. Syst. 213, 107649. http://dx.doi.org/10.1016/j.ress.2021.107649.

Univesity of Buenos Aires, 2006. VTS: Visual timed event scenarios. URL https://lafhis.dc.uba.ar/dependex/vts/.

Vyatkin, V., 2007. Model-checkers for net condition/event systems. URL http://vyatkin.org/tools/modelchekers.html.

Vyatkin, V., Bouzon, G., 1994. Timing diagrams as visual specifications in verification of industrial automation controllers. Eurasip J. Embed. Syst. 2007 (2).

Weizmann Institute of Science, 2020. PlayGo. URL https://wiki.weizmann.ac.il/playgo.

Yoo, J.-B., Cha, S.-D., Jee, E.-K., 2009. Verification of PLC programs written in FBD with VIS. Nucl. Eng. Technol. 41 (1), 79–90. http://dx.doi.org/10.5516/NET.2009.41.1.079.

Zhang, P., Li, B., Grunske, L., 2010. Timed property sequence chart. J. Syst. Softw. 83 (3), 371–390. http://dx.doi.org/10.1016/j.jss.2009.09.013.

Zhang, N., Yu, B., Tian, C., Duan, Z., Yuan, X., 2021. Temporal logic specification mining of programs. Theoret. Comput. Sci. 857, 29–42. http://dx.doi.org/10.1016/j.tcs.2020.12.032.

Antti Pakonen received the M.Sc. (Tech.) degree in I&C systems from the Helsinki University of Technology (now Aalto University), Espoo, Finland, in 2004. He is currently a Senior Scientist and a Project Manager with VTT Technical Research Centre of Finland Ltd., Espoo, where he has been employed since 2002. His research interests include I&C software engineering, model checking, I&C architecture evaluation, and knowledge management. He has applied model checking in practical industry projects in the Finnish nuclear and railway industries since 2008.

Igor Buzhinsky received the B.Sc. and M.Sc. degrees in applied mathematics and computer science from ITMO University (St. Petersburg, Russia) in 2013 and 2015, the M.Sc. degree in software engineering and service design from the University of Jyväskylä (Jyväskylä, Finland) in 2015, and the D.Sc. (Tech.) degree from Aalto University (Espoo, Finland) in 2019. His research was centered around ensuring reliability of software systems, especially with model checking and formal synthesis. He is currently an AI Developer with IPRally Technologies Oy (Helsinki, Finland).

Valeriy Vyatkin received the Ph.D. and Dr.Sc. degrees in applied computer science from Taganrog Radio Engineering Institute, Taganrog, Russia, in 1992 and 1999, respectively, the Dr.Eng. degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1999, and the Habilitation degree from the Ministry of Science and Technology of Sachsen-Anhalt, in 2002. He is on joint appointment as the Chair of Dependable Computations and Communications, Luleå University of Technology, Luleå, Sweden, and Professor of Information Technology in Automation, Aalto University, Finland. His research interests include software engineering for industrial I&C systems, artificial intelligence, distributed architectures and multiagent systems.