| | |
|---|---|
| Title | User-friendly formal specification languages - conclusions drawn from industrial experience on model checking |
| Author(s) | Pakonen, Antti; Pang, Cheng; Buzhinsky, Igor; Vyatkin, Valeriy |
| Citation | 21th IEEE Conference on Emerging Technologies and Factory Automation, ETFA 2016, 6 - 9 September 2016, Berlin, Germany |
| Date | 2016 |
| Rights | © 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. |

# User-friendly formal specification languages – conclusions drawn from industrial experience on model checking

Antti Pakonen
VTT Technical Research Centre of Finland Ltd.
Espoo, Finland
antti.pakonen@vtt.fi

Cheng Pang[1], Igor Buzhinsky[1,2], Valeriy Vyatkin[1]
[1]Department of Electrical Engineering and Automation, Aalto University
Espoo, Finland
[2]Computer Technologies Laboratory, ITMO University
St. Petersburg, Russia
cheng.pang.phd@ieee.org, igor.buzhinskii@aalto.fi, vyatkin@ieee.org

***Abstract*—Formal methods – such as model checking – have definite advantages over more commonplace verification techniques. By providing proof of the analyzed systems' correctness, they are especially useful in domains that are under regulatory supervision, like the nuclear industry. The foremost challenge for wider adoption of model checking is the effort and the expertise required for formalizing functional requirements into verifiable properties. A particular challenge in verifying the application software of industrial process control systems is taking into account the different sequencing and timing issues that arise from, e.g., the dynamic behavior of the plant processes being controlled. In this paper, we review specification languages that are aimed at making formal methods more accessible. We have collected 1079 sample formal properties from practical model checking projects in the nuclear industry, and identified repeatedly occurring property types. We present our findings, and based on the sample data, evaluate the applicability of different approaches on user-friendly property specification.**

***Keywords—model checking; formal specification languages; requirement patterns; requirements engineering; nuclear power***

## I. INTRODUCTION

Reliability of industrial instrumentation and control (I&C) systems is a question that is sometimes critical to safety and almost always critical to cost. Due to the complexity of modern digital I&C systems, exhaustive verification through conventional methods (e.g., testing, simulation, or reviews) is nevertheless practically impossible. Formal methods, on the other hand, can be used to systematically demonstrate the consistency and completeness of system specifications, and moreover, to verify the system implementation using rigorous analysis techniques and mathematical proofs [1].

The nuclear industry is one example of a domain where safety is paramount, and accordingly, the regulatory requirements are strict. It should therefore be no surprise that the nuclear industry – despite being traditionally conservative when it comes to applying emerging techniques – has adopted formal verification methods to practical use. In Finland, VTT has performed model checking of nuclear I&C software design in practice since the year 2008 [2].

The major catch is that formalization of system requirements into automatically verifiable properties is hard and error-prone [3]. The use of formalisms such as temporal logics requires a level of sophistication many users never develop [4]. The training required to reach proficiency may outweigh expected benefits [1], which presents an impediment to moving formal techniques from theoretical research to practice [5]. If used inappropriately, by insufficiently trained personnel, or without proper tool support, formal methods can even be dangerous [1].

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 language already used in the industry.

The contribution of the paper is threefold. First, after introducing model checking in the context of industrial I&C systems in Section II, we survey different approaches on user-friendly property specification in Section III. Second, in Section IV, we analyze our sample data, which is composed of 1079 collected sample properties from practical customer projects VTT has carried out in the nuclear industry. Finally, in Section V, we examine the applicability of the different user-friendly approaches based on the sample data. Our conclusions are stated in Section VI.

## II. FORMAL VERIFICATION OF INDUSTRIAL CONTROL SOFTWARE

### A. Model checking of process control application software

Model checking [6] is a formal verification method than can be used to show that a model of a (hardware or software) system fulfills specified properties. A software tool called a model checker is used to automatically determine if a property holds, taking into account all the possible states or executions

This work has been funded by the Finnish Research Programme on Nuclear Power Plant Safety 2015–2018 (SAFIR 2018), and also supported by the Government of Russian Federation, Grant 074-U01.

of the model. If an execution path contrary to a stated property is found, it is returned to the user as a counterexample.

Model checking of industrial programmable logic controller (PLC) software is a topic addressed by several authors using a range of different methods [7, 8, 9, 10]. Application areas are as diverse as those of PLCs – nuclear [2], chemical [9], rail traffic [11], automotive industry [12], or even knitting [13], just to name a few. Much research has focused on generating the system model automatically from standard PCL programming languages [14]. Even with non-standard vendor-specific languages in use, creating or generating the system model is relatively easy [14] when compared with specification of the properties. Nevertheless, model checking is still far from common industry practice [12].

In model checking, the system is usually modelled as a type of finite state machine (FSM), using a discrete model of time. For analyzing I&C systems, discrete time is usually a sufficient approximation for the cyclic processing of time in, for example, PLC equipment [15, 16]. Still, different modelling tricks are sometimes required to properly handle timed aspects of the logic [14, 17]. Techniques for real-time model checking do exist, but our experience has shown that real-time model checkers are not – as of yet – as practical in I&C applications as discrete-time tools are [18]. Other abstractions are also needed in defining the FSM in order to handle analogue data, complex algebra, and asynchrony in distributed systems [2, 14, 15]. When aiming for safety demonstration, the limitations imposed by these abstractions should be understood and documented [1].

### *B. Property formalization*

The formalized properties are typically divided in two main types. *Safety* properties state that something ("bad") should never happen, while *liveness* properties state that something ("good") must eventually happen [19]. Verification of safety properties, in particular, highlights the advantages of formal methods, as it can be difficult to assess what kind of test coverage, for example, is sufficient for demonstrating that an unwanted situation will *never* occur.

In model checking, *temporal logics* are typically used to specify the properties. A temporal logic is a formalism that describes sequences of transitions between states in a system [6]. Languages such as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) use *temporal operators*, which can be combined with Boolean logic or nested arbitrarily [6]. LTL is based on linear time, while CTL handles time as a branching structure, utilizing *path quantifiers*.

The most common LTL temporal operators are introduced in Table I. (Different notations are in use, the symbols used here are taken from [6].) In Table I, *p* and *q* are placeholders for *propositions* – statements about the state of affairs (in terms of model variable values) than are either true or false at any given time. Temporal operators can accept not only propositions, but also other temporal formulas. As for CTL temporal operators, they are obtained from LTL ones by adding path quantifiers E ("there is a path from the current state") or A ("for all paths from the current state"), e.g. EX or AX.

TABLE I. BASIC TEMPORAL OPERATORS IN LTL

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

Notably, the operators listed in Table I are all related to future states of systems, while many interesting properties of systems are nonetheless more naturally formulated in a way that addresses past states. Accordingly, temporal logics with past operators have also been developed. In Table II, we list past temporal operators developed for LTL (taken from [20]):

TABLE II. PAST TEMPORAL OPERATORS FOR LTL

| Op. | Semantics |
|---|---|
| $Y\,p$ | ("yesterday"): $p$ holds in the previous state of the path. |
| $H\,p$ | ("historically"): $p$ is true at every preceding (and the current) state on the path. |
| $O\,p$ | ("once"): $p$ is true ar some past (or the current) state on path. |
| $p\,S\,q$ | ("since"): $q$ is true at some past state, and for every state that has then followed on the path, $p$ has been true. |

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* [20].

In specifying functional requirements (and therefore also verifiable properties) for process control systems, there is a specific need to address sequences in time, mostly due to the dynamic behavior of the physical plant 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 at the alarm limit) 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, and protect the equipment.
- Feedback from 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 represent the cyclic processing of PLCs. For more exact handling of time, real-time specification languages such as MTL and TCTL have been developed (see, e.g., [21]).

## III. User-friendly specification languages

### A. 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 [22] and [23], 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*).

In [24] and [25], Campos et al. propose a collection of domain-specific specification patterns for automated production systems, based on property data collected from literature. In [26], Monteiro et al. propose a set of patterns for cellular interaction networks.

### B. Visual languages

When looking for a specification language that would be both 1) generally understandable to users not familiar with temporal logic, and 2) suited for expressing complex sequences, an obvious starting point is to look into visual languages. Here, we survey a few different approaches.

In Graphical Interval Logic (GIL) [27], 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 (see Fig. 1 for an example).

The TimeLine Editor [28] 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 come in three types: *regular* events (that do not always need to occur but make a property relevant when they do), *required* events (that must occur in response to regular events), and *fail* evens (that should not occur). Actual timing is not addressed (even in approximate terms), only the ordering of events.

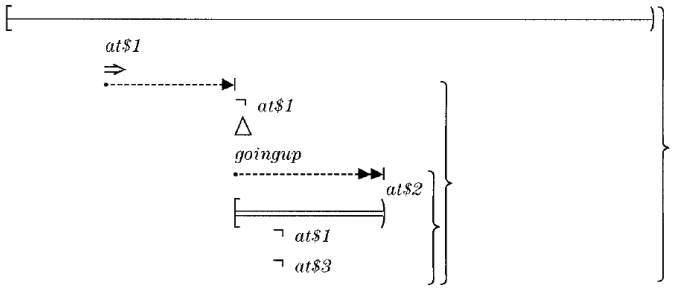


Fig. 1. A GIL specification example that reads: "The elevator goes up when it departs the first floor, arriving at the second floor without first visiting any other floors."

A visual language developed for a tool called Timing Diagram Editor (TDE) is introduced in [29]. The timing diagram consists of two parts, with the upper "if" part representing "input" signals, and the lower "then" part representing desired "outputs". When the "if" part is achieved, it is required that the changes of the "then" signals do not violate the *partial ordering* specified in the diagram. Partial ordering means that ordering between events is *not* determined by their horizontal position on the time line. In order to constrain ordering, *precedence operators* are used. While *sequential ordering* would be more intuitive, it would mean that only a single scenario would be described in each diagram. To illustrate, in Fig. 2, there are several possible scenarios, since the diagram (a) does *not* state that the rising edge of the s2 signal occurs before the rising edge of s1, until precedence operators are added (as in diagram (b)).

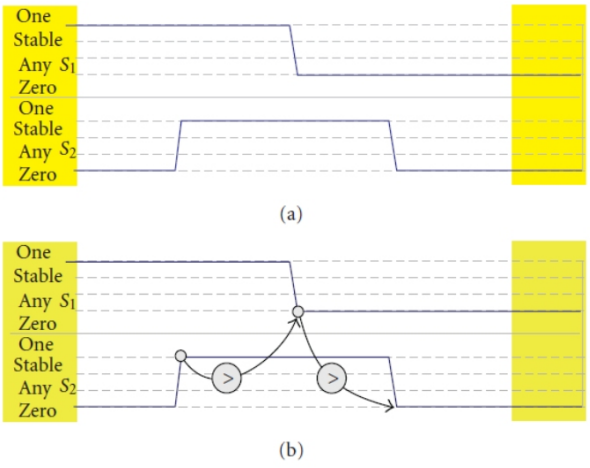


Fig. 2. In TDE specifications, the ordering of events is not specified by their horizontal position on the timeline, but by the use of precendence operators. [29]

The specification language for the TDE is reminiscent of Symbolic Timing Diagrams (STD), in development since 1993 [30].

Property Sequence Chart (PSC) [5] 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. PSC also supports partial ordering.

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. The exemplar PSC property in Fig. 3 states: "if the user has logged in and if the withdraw request has been satisfied, the bank database must be updated; the withdraw request is allowed only if the user has not logged out". Symbols on arrows denote constraints, with the filled circle in Fig. 3 standing for unwanted messages.

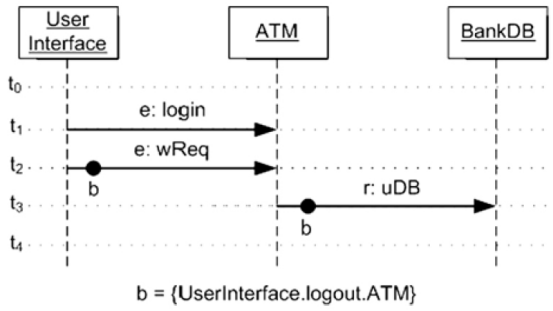


Fig. 3. An exemplar PSC property [5]

Live Sequence Charts (LSC) [31] represent another attempt at modifying UML charts in order to capture properties. LSC scenarios are described in terms of charts, locations, messages, and conditions – all of which can be "hot" (mandatory) or "cold" (provisional). Both liveness ("yes-stories") and safety ("no-stories") properties can be specified.

VTS [32] 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 on the arrows stand for forbidden events between the connected points. Temporal constraints can also be included in the arrows. A simple example of the notation can be seen in Fig. 4. The pattern in the figure depicts an execution that is true if and only if it contains a stimulus *e* that is followed by two responses, *r1* and *r2*, and there is no event *r3* between them.

In $L_R$ [33], properties are visualized as directed acyclic graphs. Nodes of the graph represent predicates (propositions), logical connections (conjunction, disjunction, negation), and temporal operators. Path quantifiers are also used. Predicates that specify inputs end with "!", while predicates that specify outputs end with "?". 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, while the matching formal template is specified manually. For an example of $L_R$, see Fig. 5.

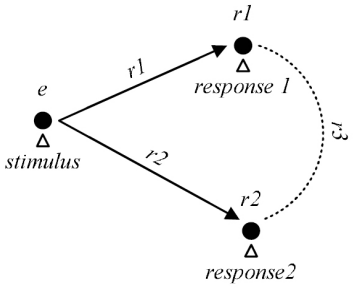


Fig. 4. An exemplar simple VTS pattern [32]

The Simulink Design Verifier [34] uses function block diagrams for specifying properties. Three basic temporal operator blocks are provided: *Detector* (for detecting and constructing inputs and outputs 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* (capturing the within implication by observing whether the second input is true for at least one time step within each true duration of the first input).

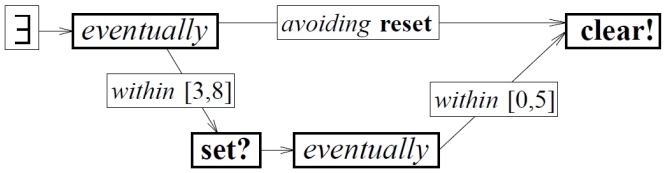


Fig. 5. An exemplar property in $L_R$ [33]

The applicability of the languages listed above to fit our domain-specific needs is discussed in Section V. Visual languages dealing with real time properties (see, e.g., [35]), are outside of our scope (with the exception of VTS), since our experience has shown that real time model checkers are so far not efficient enough for functional verification of I&C software in practice [18].

### C. PSL

Property Specification Language (PSL) [36] is an IEC and IEEE standard for the specification of properties or assertions about hardware design. It is designed to be compatible with hardware description languages (GFL, VHDL, Verilog and SystemC) via different "flavors", but can just as well be used in any context where LTL or CTL applies. Indeed, PSL is formally an extension of LTL and CTL.

The aim is that PSL specifications should be "human readable" [37]. 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 [37]:

*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 semi-colons. Here is a simple example from [37]:

*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.

## IV. Analysis of Customer Specifications

### A. Practical industry applications

Since 2008, VTT has been applying model checking in practical customer projects in the Finnish nuclear industry. On commission from the Finnish Radiation and Nuclear Safety Authority (STUK), VTT has evaluated the Protection System (PS) and the Priority and Actuation Control System (PACS) of the Olkiluoto 3 nuclear power plant (NPP) under construction. On commission from Fortum Power and Heat, VTT has performed independent, third-party verification of application functions related to, e.g., reactor protection, in support of the Loviisa NPP automation renewal projects LARA [2] and ELSA. For Fennovoima, the future operator of the NPP in Hanhikivi, VTT has evaluated the functional architecture of I&C systems.

To evaluate the applicability of different approaches on user-friendly property specification, we have collected properties written and verified in practical customer projects VTT has performed for four clients in the nuclear industry between the years 2014 and 2016. The properties have been written by two analysts, each with approximately eight years of experience in practical work. The properties have been written in LTL, CTL and PSL, to the extent which PSL is supported by the NuSMV [38] model checker.

The actual verification has been performed using the graphical modelling tool MODCHK [14], developed by VTT, using NuSMV. The models employ a manually specified function block library (since the source material is based on non-standard, vendor-specific, black-box function blocks) [14], and a discrete time modelling approach for time-dependent components [15]. The specified properties typically relate to (safety) I&C functions, expressed as function block diagrams.

As far as we are aware, no similar study has been conducted in the I&C domain. In [24] and [25], the domain-specific study was based on literature, not actual industrial applications. In [12], 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.

### B. The sample data

1079 properties were collected. 943 (87 %) of the properties were written in LTL, 133 (12 %) in PSL, and only 3 in CTL. For all the LTL properties written, the occurrence of different temporal operators is shown in Table III.

All of the PSL properties use SEREs, and specifically, 130 (98 %) of the PSL properties contain at least one SERE that employs the repetition operator [*n].

993 (92 %) of all properties can be classified as implications (due to the use of operators ->, |->, and |=>).

TABLE III. The use of temporal operators in the LTL properties

| Operator | no. of properties | of all LTL properties |
|---|---|---|
| *G* | 938 | 99.5 % |
| *X* | 122 | 12.9 % |
| *F* | 49 | 5.2 % |
| *O* | 25 | 2.7 % |
| *H* | 11 | 1.2 % |
| *Y* | 11 | 1.2 % |
| *U* | 8 | 0.8 % |

Regarding timing, of all properties, 263 (24.4 %) 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 IV, we list the property types that are used more than two times in total, and appear in the context of more than one model (I&C function).

TABLE IV. Most common types of properties in the sample data

| # | Property type | Lang. | Σ | % |
|---|---|---|---|---|
| 1 | *G* (*p* -> *q*); | LTL | 646 | 59.9 |
| 2 | *always* {*SERE*} |-> {*SERE*}!; | PSL | 102 | 9.5 |
| 3 | *G* (*p* & *X q* -> *X r*); | LTL | 64 | 5.9 |
| 4 | *G* (*p*); | LTL | 50 | 4.6 |
| 5 | *G* (*p* -> *F q*); | LTL | 31 | 2.9 |
| 6 | *G* (!*p*); | LTL | 23 | 2.1 |
| 7 | *G* (*p*) -> *G* (*q* -> *r*); | LTL | 22 | 2.0 |
| 8 | *never* {*SERE*}; | PSL | 22 | 2.0 |
| 9 | *G* (*p* -> *O q*); | LTL | 20 | 1.9 |
| 10 | *always* {*SERE*} |=> {*SERE*}!; | PSL | 9 | 0.8 |
| 11 | *G* (*p* & *X q* -> *r*); | LTL | 6 | 0.6 |
| 12 | *G* (*p*) -> *G* (*q*); | LTL | 5 | 0.5 |
| 13 | *G* (*p* -> *X q*); | LTL | 4 | 0.4 |
| 14 | *G* (*p* & *X q* & *X X r* -> *X X s*); | LTL | 3 | 0.3 |
| | Others | * | 72 | 6.7 |
| | **Total** | * | **1079** | **100** |

The temporal logic behind property types (1), (4), and (6) is the same, as there are only differences in the Boolean algebra. For the purposes of evaluating different approaches, we nevertheless separate them here. The type *G* (*p*) therefore

actually covers 719 (66.6 %) of all properties. The PSL types (2), (8) and (10), on the other hand, contain different temporal structures within each type depending on the use of SEREs.

Out of the 72 (LTL and CTL) properties categorized as others, 10 could easily be rewritten using PSL types (2), (8) or (10), but the analyst who was involved was more comfortable with nested LTL expressions. Many others could also be rewritten in PSL, although not directly with the constructs listed in Table IV. In all, none of the property types categorized as others generally repeat, nor are used in the context of more than one model. It is also worth noting that not all of the properties necessarily relate to actual functional system requirements, but may represent an attempt of the analyst to understand some aspect of the system (model) behavior.

Since the occurrence of property types (6) and (8) adds up to only 4.2 %, it would seem there are surprisingly few safety properties. However, when we consider, e.g., spurious actuation, a safety property: "the (safety) function shall not be actuated unless the process criteria is true" can be written as: *G* (*actuation -> criteria*). While this is equivalent with *G* !(*actuation* & !*criteria*), it seems to take the form of a liveness property. The former is easier to write, and also modify to the property type (9) to state: "the (safety) function shall not be actuated unless the process criteria has been true at some point in the past".

Types (7) and (12) are examples of 41 different properties (3.8 % of sample data) that begin with the expression "*G* (*p*) ->". In 35 of these 41 properties (as in (7) or (12)), what then follows is one of the other property types listed in Table IV. Reasons for using such an expression might include 1) filtering out irrelevant execution paths, 2) checking to see if a different counterexample is possible, or 3) wanting to understand how the system (model) behaves under certain assumptions.

## V. Applicability of the Specification Languages

### A. Specification patterns

Dwyer et al. claim that in their experience, 92% of collected requirements match their specification patterns, the most popular being Response ("*p* leads to *s*"), Universality ("*p* is true"), and Absence ("*p* is false") [23]. Based on Table IV, of the properties we have collected, 64.5 % ((1) and (4) combined) fall under Universality, 2.9 % under Response, and 2.1 % under Absence. The type (9) could also be captured using the Precedence pattern, bringing the total up to 71.4 %, with only four patterns. The domain-specific patterns in [24] also contain our type (13) in the form of the Immediate Response pattern, which would bring a marginal advantage of further 0.4 %. The numbers might be different if the analysts behind the case data had made a specific effort to apply the patterns, but our experience has shown that it is difficult to express I&C specific properties related to timing and sequences by using the scopes of Dwyer et al. [39].

From Table IV, we can see that just eleven LTL templates or patterns ((1), (3), (4), (5), (6), (7), (9), (11), (12), (13) and (14)) would be enough to capture 81.0 % of all sampled properties. ((1), (4), and (6) could also be combined to a more generic pattern, since their temporal aspects are equivalent.) However, 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. The relatively large share of the PSL properties 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 [26].

Still, if we were to introduce new, domain-specific specification patterns for I&C (similarly as authors of [24], as well as [26] in their own field, have done), an obvious candidate would be the third-most common property type in Table IV: *G* (*p* & *X q -> X r*), or: "*p* followed on the next time step by *q* shall lead to *r*" (5.9 % of collected properties). An exemplar application of this pattern is to state: *G* (!*p* & *X p -> X q*), or: "a rising edge of *p* shall lead to *q*".

### B. Visual languages

In graphical specification languages based on sequence diagrams, properties are seen as relations on a set of messages being exchanged between system components, with different constraints applied (messages being mandatory, provisional, etc.) [5]. Sequence diagrams tend to depict (wanted or unwanted) *scenarios*, while verifiable properties should focus on transitions between system states in a reactive system not designed to terminate [6]. Safety properties are particularly challenging, since "anti-scenarios" can easily lead to cumbersome and error-prone diagrams [32].

Graphical languages based on timing diagrams, where states of variables are visualized on a time line, are undoubtedly familiar to many engineers, but additional elements are often needed for expressing constraints. Effective property formalization calls for (seemingly counterintuitive) partial ordering, where the horizontal position of events on a time line, for example, does *not* necessarily determine the order in which events have to take place. Sequential ordering – while common in our sample data – as an *only* option limits the representational capabilities [29], and "we sometimes want to specify that the relative occurrence times between certain events are irrelevant" [3].

Instead of analyzing in detail how each individual graphical language can be used to capture our sample properties, we can identify general, I&C domain-specific criteria:

- **C1: Modelling of time** should be possible, at least in discrete terms.
- **C2: Not limiting to message exchange** – The "events" of VTS, for example, are a more versatile starting point than messages, and closer to the idea of propositions referring to system (model) states.
- **C3: Resemblance to a language already used in I&C engineering** would ease the adoption by industry experts.
- **C4: Availability of tools** has obvious benefits, but also suggests that the approach is applicable in practice.

- **C5: Partial ordering** should be supported (see above).
- **C6: Alternative state paths** (such as: "A leads to B *or* C") should be supported.

In Table V, we list visual languages mentioned in Section III, to see how they meet the above criteria.

Based on Table V, it would seem that the Simulink Design Verifier holds the most potential. Still, the expressiveness comes at a cost, as the analyst still has to get familiar with temporal logic operators, here in the form of specialized function blocks. Furthermore, using the same language for stating both the requirements and the design can lead to confusion, if, for example, temporal operator blocks are confused with delay blocks.

TABLE V. CRITERIA FOR VISUAL SPECIFICATION LANGUAGES

| | C1 | C2 | C3 | C4 | C5 | C6 |
|---|---|---|---|---|---|---|
| Simulink Design Verifier | + | + | + | + | + | + |
| Graphical Interval Logic | | + | | | + | + |
| $L_R$ | + | + | | | + | + |
| TimeLine Editor | | + | | [a] | | |
| VTS | + | + | | [b] | + | |
| PSC | | | | +[c] | | + |
| LSC | | | | +[d] | + | |
| STD | [e] | + | + | [f] | + | |
| Timing Diagram Editor | | + | + | [g] | + | |

[a.] Has been available at some point from Bell Labs?

[b.] A Microsoft Visio plugin is described in [32], but it is apparently unavailable.

[c.] CHARMY [5], PragmaDev Tracer (http://www.pragmadev.com/product/tracing.html)

[d.] Play-Engine (http://www.wisdom.weizmann.ac.il/~playbook/)

[e.] "Distance measures" are a part of an extension of STD called STDx [30].

[f.] CheckOff is introduced in [30], but is apparently unavailable.

[g.] Source code for the TDE tool [29] is in theory available, but the implementation is over ten years old and left at a prototypical stage.

### C. PSL

PSL is a textual language, but it does meet all the requirements we specified for visual specification languages in Table V. It is especially convenient in expressing sequences and (discrete) timing through the use of SEREs. For analysts already familiar with temporal logic, PSL is – as intended – a user-friendly specification language.

Looking at our sample data, all the LTL property types listed in Table IV could easily be expressed using PSL, with the exception of (9), which contains the past operator *O*. This formula is useful in expressing safety properties of the type: "the function *p* should only be activated, if there was a criteria *q* active in some point in the past". The formula can be replaced with the equivalent [40] PSL expression !(!*q until* (*p* & !*q*)), or based on the "Precedence" [23] pattern: (*always* !*p*) | (!*p until q*), but both formulae are nevertheless less convenient to read or write.

In all, there are only three types of PSL property types in Table IV, and within those types, the only aspect that varies is how the SEREs are constructed. A visual language (and a tool) based on easy manipulation of SERE-type properties would therefore seem like a worthwhile topic for further research.

## VI. CONCLUSIONS

The sample data we have collected from VTT's practical customer work on model checking supports the conclusions others have already drawn: a relatively small number of specification 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 application software design needs also to support properties that deal with timing issues and sequences: 24.4 % of all sampled properties referred to certain specific future or past states.

Many of the attempts at a visual specification language are based on representations I&C professionals are already familiar with, such as sequence or timing diagrams. However, such representations tend to visualize single (wanted or unwanted) *scenarios*, while there are a practically infinite number of scenarios than can fulfill or violate a stated *property*. To express properties, the languages have to introduce features and constructs that are unfamiliar, or even counterintuitive. The right balance between expressiveness and readability is hard to achieve, and the current lack of established and maintained tools speaks for itself.

If we were to present the user with a set of nine to eleven LTL patterns (whose temporal structure would not have to be modified in use, only the Boolean algebra inside the propositions) *and* a graphical, user-friendly way of specifying properties that conform to the basic SERE-type PSL implications, we would have covered 93.3 % of the properties in the sample data. Knowing that many of the properties belonging to the remaining 6.7 % 1) could in many cases be modified to fit given patterns, and/or 2) are likely not all that essential for verifying given functional requirements (but rather, e.g., the analyst trying understand some aspect of the system (model) behavior), these numbers seem satisfactory.

PSL is a good option for users already familiar with temporal logic. Given our sample data, the only noteworthy disadvantage of PSL is the lack of past operators. The formula "*G* (*p* -> *O q*)", specifically, is quite useful for expressing certain safety properties. The equivalent formulae without the past operator *O* are more difficult to read. The formula is used in 1.9 % of properties in the sample data.

Finding a user-friendly formal specification language has been a research topic for over twenty years. Had the problem been already solved, formal methods such as model checking would today be in much wider use.

## ACKNOWLEDGMENT

The authors wish to thank VTT's clients in the nuclear industry for permitting the use of customer data for research purposes.

## REFERENCES

[1] BEL V, BfS, CNSC, CSN, ISTec, ONR, SSM, and STUK, "Licensing of safety critical software for nuclear reactors – Common position of

international nuclear regulators and authorised technical support organisations," Revision 2014.

[2] A. Pakonen, J. Valkonen, S. Matinaho, and M. Hartikainen, "Model Checking for Licensing Support in the Finnish Nuclear Industry," International Symposium on Future I&C for Nuclear Power Plants (ISOFIC 2004), Jeju, Republic of Korea, August 24-28, 2014.

[3] R. Schlör, B. Josko, and D. Werth, "Using a visual formalism for design verification in industrial environments," Lecture Notes in Computer Science, vol. 1385, pp. 208-221, 1998.

[4] G. J. Holzmann, "The logic of bugs," Proceedings of the 10th SCM SIGSOFT symposium on Foundations of software engineering, pp. 81-87, ACM, New York, 2002.

[5] M. Autili, P. Inverardi, and P. Pellicione, "Graphical scenarios for specifying temporal properties: an automated approach," Automated Software Engineering, vol. 14, pp. 293-340, 2007.

[6] E. M. Clarke, O. Grumberg, and D. Peled, Model Checking. The MIT Press, 1999.

[7] J. Thieme and H. M. Hanisch, "Model-based generation of modular PLC code using IEC61131 function blocks," Proceedings of the 2002 IEEE International Symposium on Industrial Electronics (ISIE 2002), vol. 1, pp. 199-204, 2002.

[8] H. M. Hanisch, A. Lobov, J. L. Martinez Lastra, R. Tuokko, and V. Vyatkin, "Formal Validation of Intelligent Automated Production Systems towards Industrial Applications", International Journal of Manufacturing Technology and Management, vol. 8, pp. 75-106, 2006.

[9] B. Schlich, J. Brayer, J. Wernerus, and S. Kowalewski, "Direct Model Checking of PLC Programs in IL," 2nd IFAC Workshop on Dependable Control of Discrete Systems, Bari, Italy, 2009.

[10] V. Gourcuff, O. De Smet, and J. Faure, "Efficient representation for formal verification of PLC programs," 8th International Workshop on Discretet Event Systems, Ann Arbor, Michigan, USA, 2006, pp. 182-187.

[11] O. Pavlovic and H. Ehrich, "Model Checking PLC Software Written in Function Block Diagram," 2010 Third International Conference on Software Testing, Verification, and Validation (ICST 2010), Paris, France, 2010, pp. 439-448.

[12] O. Ljungkranz, K. Åkesson, M. Fabian, A. H. Ebrahimi, "An empirical study of control logic specifications for programmable logic controllers," Empirical Software Engineering, vol. 19, pp. 655-677, 2014.

[13] T. Reinbacher, M. Horauer, B. Schlich, J. Brauer, and F. Scheuer, "Model checking embedded software of an industrial knitting machine," International Journal of Information Technology, Communications and Convergence, vol. 1, pp. 186-205, 2011.

[14] A. Pakonen, T. Mätäsniemi, J. Lahtinen, and T. Karhela, "A Toolset for model checking of PLC software," 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA2013), 2013.

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

[16] O. Ljungkrantz, K. Åkesson, M. Fabian, and Y. Chengyin, "A Formal Specification Language for PLC-based Control Logic," 8th IEEE International Conference on Industrial Informatics (INDIN 2010), pp. 1067-1072, 2010.

[17] J. Yoo, S. Cha, and E. Jee, "Verification of PLC programs written in FBD with VIS," Nuclear Engineering and Technology, vol. 41, pp. 79-90, 2009.

[18] J. Lahtinen, K. Björkman, J. Valkonen, J. Frits, and I. Niemelä, "Analysis of an emergency diesel generator control system by compositional model checking – MODSAFE 2010 Work Report," VTT Working Papers 156, VTT, 2010.

[19] L. Lamport, "Proving the correctness of multiprocess programs," IEEE Transactions on Sofware Engineering, vol. SE-3, pp. 125-143, 1977.

[20] M. Benedetti and A. Cimatti, "Bounded Model Checking for Past LTL," Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003), LNCS, vol. 2619, pp. 18-33, 2003.

[21] P. Bouyer, "Model-checking timed temporal logics," 5th Workshop on Methods for Modalities (M4M5), Elec. Notes Theo. Comput. Sci., vol. 231, pp. 323-341, 2009.

[22] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property specification patterns for finite-state verification," Proceedings of the 2nd workshop on Formal methods in software practice (FMSP '98), ACM, New York, 1998.

[23] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-State Verification," Proceedings of the 21st International Conference on Software Enginnering (ICSE'99), ACM, New York, 1999.

[24] J. C. Campos and J. Machado, "Patter-based Analysis of Automated Production Systems," 13th IFAC Symposium on Information Control Problems in Manufacturing, Moscow, Russia, June 3-5, 2009.

[25] J. C. Campos, J. Machado, and E. Seabra, "Property Patterns for the Formal Verification of Automated Porductions Systems," 13[7th] IFAC World Congress, Seoul, Korea, 2008, pp. 5107-5112.

[26] P. T. Monteiro, D. Ropers, R. Mateescu, A. T. Freitas, and H. de Jong, "Temporal logic patterns for querying dynamic models of cellular interaction networks," Bioinformatics, vol. 24, pp. 227-233, Oxford University Press, 2008.

[27] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna, "A Graphical Interval Logic for Specifying Concurrent Systems," ACM Transactions on Software Engineering and Methodology, vol. 3., pp. 131-165, 1994.

[28] M. H. Smith, G. J. Holtzmann, and K. Etessami, "Events and constraints: a graphical editor for capturing logic properties of programs," 5th International Symposium on Requirements Engineering, Toronto, pp. 14-22, 2001.

[29] V. Vyatkin and G. Bouzon, "Timing Diagrams as Visual Specifications in Verification of Industrial Automation Controllers," EURASIP Journal of Embedded Systems, Vol. 2008, November, 2007.

[30] R. C. Shlör, "Symbolic Timing Diagrams: A Visual Formalism for Model Verification," Doctoral thesis, Fachbereich Informatik, Carl-von-Ossietzky Universität Oldenburg, Germany, 2001.

[31] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," Formal Methods in System Design, vol. 19, pp. 45-80, 2001.

[32] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero, "Visual Timed Event Scenarios," 26th International Conference on Software Engineeing (ICSE'04), pp. 168-177, 2004.

[33] I. Lee and O. Sokolsky, "A graphical property specification language," High-Assurance Systems Engineering Workshop, Washington, pp. 42-27, 1997.

[34] Simulink Design Verifier Release Notes: Available at: http://se.mathworks.com/.

[35] Y. Oh, "Visual Approaches for System Property Specification," 2003.

[36] IEC 62531:2012, IEE Std. 1850-2010, Property Specification Language (PSL), International standard, June, 2012.

[37] C. Eisner and D. Fisman, A Practical Introduction to PSL. Springer Science, 2006.

[38] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," 14th International Conference on Computer-Aided Verification (CAV 2002), pp. 359-364, 2002.

[39] T. Tommila and A. Pakonen, "Controlled natural language requirements in the design and analysis of safety critical I&C systems," VTT-R-01067-14, VTT, Espoo, 2014.

[40] M. Pradella, P. San Pietro, P. Spoletini, and A. Morzenti, "Practical model checking of LTL with Past," 1st International Workshop on Automated Technology for Verification and Analysis (ATVA03).