Metadata of the chapter that will be visualized in SpringerLink

Book Title	Fundamentals of Software Engineering
Series Title	
Chapter Title	Event-Driven Temporal Logic Pattern for Control Software Requirements Specification
Copyright Year	2021
Copyright HolderName	IFIP International Federation for Information Processing

Corresponding Author	Family Name	**Zyubin**
	Particle	
	Given Name	**Vladimir**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	Institute of Automation and Electrometry
	Address	Novosibirsk, Russia
	Email	zyubin@iae.nsk.su
Author	Family Name	**Anureev**
	Particle	
	Given Name	**Igor**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	Institute of Automation and Electrometry
	Address	Novosibirsk, Russia
	Email	anureev@iis.nsk.su
Author	Family Name	**Garanina**
	Particle	
	Given Name	**Natalia**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	Institute of Automation and Electrometry
	Address	Novosibirsk, Russia
	Email	garanina@iis.nsk.su
Author	Family Name	**Staroletov**
	Particle	
	Given Name	**Sergey**
	Prefix	
	Suffix	

	Role	
	Division	
	Organization	Institute of Automation and Electrometry
	Address	Novosibirsk, Russia
	Email	serg_soft@mail.ru
Author	Family Name	**Rozov**
	Particle	
	Given Name	**Andrei**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	Institute of Automation and Electrometry
	Address	Novosibirsk, Russia
	Email	rozov@iae.nsk.su
Author	Family Name	**Liakh**
	Particle	
	Given Name	**Tatiana**
	Prefix	
	Suffix	
	Role	
	Division	
	Organization	Institute of Automation and Electrometry
	Address	Novosibirsk, Russia
	Email	antsys_nsu@mail.ru

Abstract	This paper presents event-driven temporal logic (EDTL), a specification formalism that allows the users to describe the behavior of control software in terms of events (including timeouts) and logical operations over inputs and outputs, and therefore consider the control system as a “black box”. We propose the EDTL-based pattern that provides a simple but powerful and semantically rigorous conceptual framework oriented on industrial process plant developers in order to organize their effective interaction with the software developers and provide a seamless transition to the stages of requirement consistency checking and verification.

Event-Driven Temporal Logic Pattern for Control Software Requirements Specification

Vladimir Zyubin(✉), Igor Anureev, Natalia Garanina, Sergey Staroletov, Andrei Rozov, and Tatiana Liakh

Institute of Automation and Electrometry, Novosibirsk, Russia
{zyubin,rozov}@iae.nsk.su, {anureev,garanina}@iis.nsk.su

Abstract. This paper presents event-driven temporal logic (EDTL), a specification formalism that allows the users to describe the behavior of control software in terms of events (including timeouts) and logical operations over inputs and outputs, and therefore consider the control system as a "black box". We propose the EDTL-based pattern that provides a simple but powerful and semantically rigorous conceptual framework oriented on industrial process plant developers in order to organize their effective interaction with the software developers and provide a seamless transition to the stages of requirement consistency checking and verification. AQ1 AQ2

1 Introduction

Most current proposals that are intended to improve software quality and rely on formal methods, are rejected by the mainstream practice. Fast-moving software development companies do not consider it cost-effective to apply such methods in their software development processes, because the critical issue in the field is not quality but rather the "time-to-market" [1].

The situation is different in industrial programming. This includes PLC-based control systems, embedded systems, and such present-day initiatives as cyber-physical systems, and Industrial Internet of Things, where emergent system properties such as safety, correctness, robustness, and maintainability are very important [2]. This enforces developers of such safety-critical software to use formal methods. However as the size of systems grows, expenses that are required to use formal methods, grow disproportionately. Hence, these methods can only be applied to relatively small systems [3].

Another circumstance causing formal methods to be expensive in this domain is their conceptual discrepancy with the specifics of industrial plant engineering.

This work has been funded by the state budget of the Russian Federation (IA&E project No. AAAA-A19-119120290056-0). Authors are very grateful for the charitable support they received from the JetBrains Foundation.

H. Hojjat and M. Massink (Eds.): FSEN 2021, LNCS 12818, pp. 1–16, 2021.
https://doi.org/10.1007/978-3-030-89247-0_7

Modern studies show that this problem is actually very acute to this date [4]. Control software development fits well into the "client-contractor" paradigm. At the initial stages of the project (system requirements specification, program specification), the client plays a leading and irremovable role. Their input gradually decreases as the project progresses to the implementation stage. The contractor (programmer), however, plays an auxiliary and dependent role at the start. It is only at the design and implementation stages of the project that they start to gain relative independence.

The main contradictions we face here are the following: (a) the clients think in terms of events, timeouts, processes and states [5], while the contractors are limited to the programming languages they use, e.g. the IEC 61131-3 languages [6], (b) the clients do not bother seeing into the internal structure of control software, whereas the contractors neglect learning the inherent principles of processes within the plant, (c) the plant is designed by the clients and, as an artifact, it already implicitly assumes a control algorithm by design, yet the contractors need to specify this hidden algorithm in a strict form.

This explains why most bugs in critical systems are a result of incompleteness or other flaws in the software requirements, not coding errors [7]. This also means that we should focus not on requirement checking, but rather on how to formulate a complete and correct set of requirements, and further check them for consistency.

The following attempts to solve this problem are known: using a pattern-restricted natural language [8–11], using information extraction methods to get the necessary information from natural language specifications [12–14], using domain-oriented (FSM-based) languages [15], using graphic notations [16], formal requirement pattern languages [17–23], to mention a few.

Summarizing the above, we can formulate the general principles of requirements specification for control software. A requirements specification should be:

- *user-friendly*, i.e. correspondent to the process plant design and based on the concepts of events (including timeout events) and reactions;
- *independent of control software design and implementation*, that is, it should use the black box principle and operate in terms of inputs and outputs, without any knowledge of the inner structure of either the control software or the plant hardware;
- following a *unified pattern*;
- *strict*, i.e. it should have formal semantics;
- *universal*, i.e. not orientated towards any particular verification technique.

In this paper, we develop such a specification and demonstrate its use with a simple but practical case study.

The rest of the paper consists of three principal parts. In Sect. 2, we propose a conceptual schema for the requirements specification and its syntax, then in Sect. 3, we construct an informal semantics of the notation. Finally, in Sect. 4, we demonstrate the proposed notation on a hand dryer control system. In Appendix A, we present our bounded checking algorithm for the proposed specifications and discuss its implementation.

2 Syntax and Definition of EDTL-Requirements

In this section, we describe the syntax of the proposed notation for requirements.

Definition 1. *(EDTL-requirements)*
An EDTL requirement is a tuple of the following attributes:

$$R = (\mathbf{trigger}, \mathbf{invariant}, \mathbf{final}, \mathbf{delay}, \mathbf{reaction}, \mathbf{release}).$$

The graphical intuition for the temporal orchestration of EDTL-attributes is shown in Fig. 1. Table 1 gives the informal description of the attributes.

Table 1. The EDTL attributes

Attribute	Description
Trigger	An event after which the invariant must be true until a release event or a reaction takes place; this event is also the starting point for timeouts to produce final/release events (if any)
Invariant	A statement that must be true from the moment the trigger event occurs until the moment of a release or reaction event
Final	An event, after which a reaction must occur within the allowable delay. This event always follows the trigger event
Delay	A time limit after the final event, during which a reaction must appear
Reaction	This statement must become true within the allowable delay from the final event
Release	Upon this event, the requirement is considered satisfied

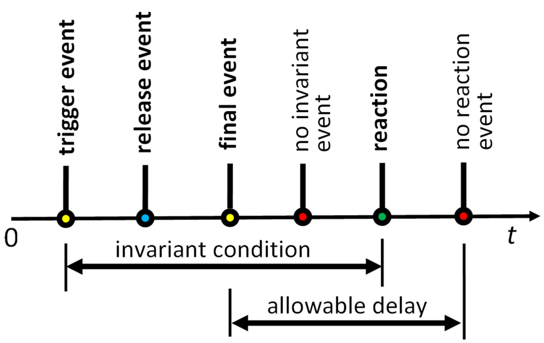

Fig. 1. Concept of a requirement specification in EDTL

The value of each attribute of EDTL-requirements is an *EDTL-formula*. This formula is a Boolean formula built from *EDTL-terms*. The EDTL-formulas are also enriched with special Boolean terms for monitoring instantaneous changes of system variables' values: *changes*, *increases*, and *decreases*. The Boolean term *passed* describes that a control system is in a state after moment specified by a term of type *time*.

Definition 2. *(EDTL-terms)*
The terms are built from typed constants, variables and functions:

- A constant of a type t is a term of the type t.
- A variable of a type t is a term of the type t.
- If $u_1, \ldots, u_n$ are terms of types $t_1, \ldots, t_n$, and f is a function of type $t_1 \times \ldots \times t_n \rightarrow t$, then $f(u_1, \ldots, u_n)$ is a term of a type t.
- If u is a term, then (u) is a term.

The set of types includes types *int* (for integers), *double* (for floating points), *bool* (for Boolean values $true$ and $false$), and *time* (e.g., $1h$ and $1s$ for 1 h and 1 s). The functions include standard arithmetic operations and relations, Boolean operations and C-like bitwise operations.

EDTL-formulas are constructed from Boolean terms by standard Boolean operations and special operations for expressing instant control system changes.

Definition 3. *(EDTL-formulas)*
If ϕ and ψ are EDTL-formulas then:

- ETDL-term of type *bool* is an atomic EDTL-formula;
- $\phi \wedge \psi$ is the conjunction of ϕ and ψ;
- $\phi \vee \psi$ is the disjunction of ϕ and ψ;
- $\neg\phi$ is the negation of ϕ;
- $\backslash\phi$ is the falling edge: the value of ϕ changes from $false$ to $true$;
- $/\phi$ is the rising edge: the value of ϕ changes from $true$ to $false$;
- $\_\phi$ is low steady-state: the value of ϕ remains equal to $false$;
- $\sim \phi$ is high steady-state: the value of ϕ remains equal to $true$.

3 Semantics of EDTL-Requirements

3.1 Definitions for the Semantics

The syntax and informal meaning of EDTL-requirements to a control system do not depend on implementations of this control system. However, we must define an abstract model of a control system to describe the formal semantics of EDTL-requirements corresponding to their intuitive understanding. To do this we will use the cyclic scan or triggered execution model defined in the IEC 61131-3 [6].

We consider that a control system functioning consists of an infinite sequence of *scan-cycles*. Each scan cycle includes a sequence of three phases: reading input, execution, and writing output. Our model of a control system [24] abstracts from scan cycle time (the environment is considered to be slow enough to assume zero time for the input/output and execution phases of a scan cycle) [25]. Hence, we give the semantics to EDTL-requirements in discrete time paradigm: values of input and output variables of a control system are observable in states at the beginning of a scan cycle. Due to the black-box principle [26], we consider that EDTL-formulas include input and output variables only. In definitions of formal semantics, we take into account that input variables are evaluated at the beginning of a scan cycle and not changed during the scan cycle, and, in contrast, output variables are changed during a scan cycle and finally evaluated at the end of the scan cycle. We consider a control system as a standard transition system:

Definition 4. *(Control systems)*
A control system is a transition system $CS = (S, I, R)$, where

- S is a set of states, and
- $I \subset S$ is a finite set of initial states, and
- $R \subseteq S \times S$ is a total transition relation.

A path $\pi = s_0, s_1, \ldots$ is an infinite sequence of states $s_i \in S$ such that $\forall j > 0 \;:\; (s_j, s_{j+1}) \in R$. In state s_i on path π, i is a number of a scan cycle (*called a time point*), and $\pi(i) = s_i$. *An initial path* π^0 is a path starting from initial state, i.e. $\pi^0(0) \in S_0$.

In EDTL-requirements, a special attention is paid to time (or event) constraints. Hence, we introduce *a timer point* which is the time point on a path to define the moment of starting a timer. Timers are used to specify timeout events. We define the value of terms on path π in the current time point i w.r.t. timer point j. The fact that a term u has the value v in a state s_i means that v is the value of u at the time moment i. For variables, the value is defined by the function *acc*: $acc(x, s)$ returns the value of the variable x in the state s. For time terms, the value is defined by the function *time*: $time(u, \pi(i))$ returns the number of scan cycles which will be passed during u time with the time point i for path π. For a function f, let $intr(f)$ be a value of f.

The function *value* defines semantics (value) of EDTL-terms at time point i on path π with timer point j:

Definition 5. *(Semantics of EDTL-terms)*

- if c is a constant, then $value(c, \pi, i, j) = c$;
- if x is a variable, then $value(x, \pi, i, j) = acc(x, \pi(i))$;
- if u is a time term, then $value(u, \pi, i, j) = time(u, \pi(i))$;
- $value(f(u_1, \ldots, u_n), \pi, i, j) = intr(f)(value(u_1, \pi, i, j), \ldots, value(u_n, \pi, i, j))$;
- $value((u), \pi, i, j) = value(u, \pi, i, j)$.
  Let u be not a term of type *time* and $i > 0$:
- $value(changes(u), \pi, i, j) = true$ iff $value(u, \pi, i-1, j) \neq value(u, \pi, i, j)$;
- $value(increases(u), \pi, i, j) = true$ iff $value(u, \pi, i-1, j) < value(u, \pi, i, j)$;
- $value(decreases(u), \pi, i, j) = true$ iff $value(u, \pi, i-1, j) > value(u, \pi, i, j)$;
  Let u be a term of type *time* and $i > 0$:
- $value(passed(u), \pi, i, j) = true$ iff $i \geq j + value(u, \pi, i, j)$, i.e. $value(u, \pi, i, j)$ time steps have passed after the timer point j.

Semantics of EDTL-formulas is defined in terms of satisfiability relation between the time point with its timer point on the path of the control system: $CS, \pi, i, j \models \phi$ iff ϕ is true at time point i w.r.t. timer point j on the path π of control system CS. In this definition, we omit the name of a control system:

Definition 6. *(Semantics of EDTL-formulas)*

- $\pi, i, j \models u$ iff u is a Boolean EDTL-term and $value(u, \pi, i, j) = true$;
- $\pi, i, j \models \phi \wedge \psi$ iff $i, j \models \phi$ and $\pi, i, j \models \psi$;
- $\pi, i, j \models \phi \wedge \psi$ iff $\pi, i, j \models \phi$ or $i, j \models \psi$;
- $\pi, i, j \models \neg\phi$ iff $\pi, i, j \not\models \phi$;
- $\pi, i, j \models /\phi$ iff $i > 0$, $\pi, i\text{-}1, j \not\models \phi$ and $\pi, i, j \models \phi$;
- $\pi, i, j \models \backslash\phi$ iff $i > 0$, $\pi, i\text{-}1, j \models \phi$ and $\pi, i, j \not\models \phi$;
- $\pi, i, j \models \sim \phi$ iff $i > 0$, $\pi, i\text{-}1, j \models \phi$ and $\pi, i, j \models \phi$;
- $\pi, i, j \models \_\phi$ iff $i > 0$, $\pi, i\text{-}1, j \not\models \phi$ and $\pi, i, j \not\models \phi$.

For every EDTL-formula ϕ, $value(\phi, \pi, i, j) = true$ iff $\pi, i, j \models \phi$.

The following natural language description of EDTL-requirement semantics corresponds to the informal description of attributes in Table 1:

> *Following each trigger event, the invariant must hold true until either a release event or a final event. The invariant must also hold true after final event till either the release event or a reaction, and besides the reaction must take place within the specified allowable delay from the final event.*

We define two kind of formal semantics for EDTL-requirements. The proof of equivalence of this two semantics is out of the scope of this paper. For EDTL-requirement tp, let $trigger$, $invariant$, $final$, $delay$, $reaction$, and $release$ be EDTL-formulas which are the values of the corresponding tp attributes.

3.2 The First Order Logic Semantics

EDTL-requirement tp is satisfied in a control system CS iff the following FOL-formula F_{tp} is true for every initial path π^0:

$$
\begin{aligned}
&F_{tp} = \forall\ \pi^0 \in CS\ \forall\ t \in [1, +\infty)(\\
&\quad value(trigger, \pi^0, t, 0) \wedge \neg value(release, \pi^0, t, t) \Rightarrow \\
&\qquad \forall f \in [t, +\infty)(\forall i \in [t, f](\neg value(release, \pi^0, m, t)) \Rightarrow \\
&\qquad\quad (\forall i \in [t, f](\neg value(final, \pi^0, i, t)) \Rightarrow \\
&\qquad\qquad \forall i \in [t, f](value(invariant, \pi^0, i, t))) \wedge \\
&\qquad\quad (\forall i \in [t, f)\ \neg value(final, \pi^0, i, t) \wedge value(final, \pi^0, f, t) \Rightarrow \\
&\qquad\qquad \forall d \in [f, +\infty)(\forall i \in [f, d]\ \neg value(release, \pi^0, i, t) \Rightarrow \\
&\qquad\qquad\quad (\forall i \in [f, d](\neg value(delay, \pi^0, i, f) \wedge \neg value(reaction, \pi^0, i, f) \Rightarrow \\
&\qquad\qquad\qquad \forall i \in [f, d](value(invariant, \pi^0, i, f))) \wedge \\
&\qquad\qquad\qquad (((f \neq d \Rightarrow \forall i \in [f, d)(\neg value(delay, \pi^0, i, f) \wedge \\
&\qquad\qquad\qquad\quad \neg value(reaction, \pi^0, i, j)) \wedge value(delay, \pi^0, d, f))) \Rightarrow \\
&\qquad\qquad\qquad\qquad value(reaction, \pi^0, d + 1, f)))).
\end{aligned}
$$

In this formula, t stands for the time point of the trigger event, f stands for the time point of the final event, and d stands for the time point when the delay is over. This semantics can be used in deductive verification of control systems w.r.t. EDTL-requirements. For this, the control system should be also represented as FOL-formula F_{CS} and the implication $F_{CS} \Rightarrow F_{tp}$ should be verified. For EDTL-requirement tp, the formula F_{tp} gives the constructive way to check tp on the given finite set of finite initial paths of control system CS. This bounded checking algorithm is described in Appendix A.

3.3 The Linear Temporal Logic Semantics

EDTL-requirement tp is satisfied in a control system CS iff the following LTL [27] formula Φ_{tp} is satisfied for every initial path π^0:

$$\Phi_{tp} = \mathbf{G}(trigger \to ((invariant \wedge \neg final \mathbf{W} release) \vee \\ (invariant \mathbf{U}(final \wedge (invariant \wedge delay \mathbf{U}(release \vee reaction)))))).$$

We use this semantics in model checking control systems w.r.t. the EDTL-requirements.

4 Case Study

The Hand dryer is a simple control system which uses a hands sensor as an input and a dryer switching device as an output. Despite the apparent simplicity, the control of the object is nontrivial due to the instability of the sensor readings caused by the movement of the hands—during the drying of hands, the sensor may indicate a short-term absence of the hands. A more detailed description of the system and the implementation of the control software can be found in [5].

Due to the blackbox principle, we abstract from the control logic and observe only the input and output values.

We formulate the following requirements:

1. If the dryer is on, then it turns off after no hands are present for 1 s.
2. If the dryer was not turned on and hands appeared, it will turn on after no more than 1 cycle.
3. If the hands are present and the dryer is on, it will not turn off.
4. If there is no hands and the dryer is not turned on, the dryer will not turn on until the hands appear.
5. The time of continuous work is no more than an hour.

The tabular form for these requirements is presented in Table 2.

To demonstrate the simplicity of using the proposed notation, we illustrate the transformation of requirements into the tabular form and back with the example of requirement R1 "If the dryer is on, then it turns off after no hands are present for 1 s" (Fig. 2).

Table 2. Tabular properties for hand dryer

Req ID	Trigger event	Release event	Final event	Allowable delay	Invariant	Reaction
R1	`\H && D`	`H`	`passed(1s)`	`passed(0.01s)`	`D`	`!D`
R2	`/H && !D`	`false`	`true`	`true`	`!D`	`D`
R3	`H && D`	`false`	`!H`	`true`	`D`	`true`
R4	`!H && !D`	`H`	`false`	`true`	`!D`	`true`
R5	`/D`	`\D`	`passed(1h)`	`true`	`true`	`\D`

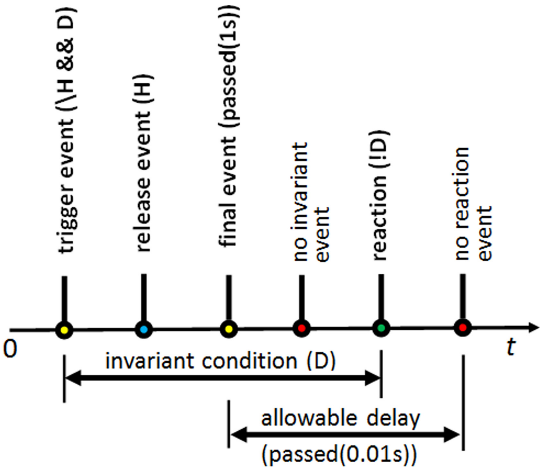

Fig. 2. Graphical representation of the requirement R1

Converting a natural-language requirement into an EDTL-record (direct transformation). The trigger event is “if the dryer is on and hand input is on falling edge”, i.e. `D && \H`. The condition “after no hands are present” means that the appearance of hands cancels the requirement checking until the next trigger event, i.e. the release event is `H`. If the new state “dryer is on and no hands” is continued, the final event “within 1 s” occurs, i.e. the final event is `passed(1s)`. The reaction to the final event is turning off the dryer (the reaction is `!D`). Since the original statement assumes that the dryer remains on until it is turned off, the invariant is `D`.

Converting an EDTL-record into a natural-language requirement (reverse transformation). The trigger event `D && \H` means the hand disappearing event when the dryer is on. The event starts checking the truth of the invariant `D` (dryer is on), until the release event `H` (the hand appearance). If the release event (`H`, or hands appear) does not occur until the final event (`passed(1s)`, during 1 s), then the control system should react (generate reaction) to this by `!D`, that is by switching off the dryer. Invariant `D` means the dryer should be on till the dryer is switched off after 1 s.

As to the allowable delay value, it can be interpreted independently from the other attributes and serves to provide the ability to specify time delays associated with execution overheads. According to the allowable delay value, the reaction should occur within 10 ms interval of time after the final event.

5 Related Work

While various models are increasingly used in the development and verification of cyber-physical systems (see our review in [2]), the development of requirements for them today stands out as a separate discipline. According to Zave [28], requirements engineering (RE) is the branch of software engineering concerned with the real-world goals for, functions of, and constraints on software systems. In its early years, requirements engineering was about the importance of specifying requirements, focusing on the 'What' instead of the 'How'. It then moved to systematic processes and methods, focusing on the 'Why' [29]. With the increasing complexity of requirements, the question on their organization arises.

Starting with IEFC RFC 2119 [30], an attempt was made to prioritize them in baseline text form claims, at the same time, English modal verbs like *"Must"*, *"Should"*, *"May"* were used as keywords for the degree of desirability of requirements. In [31] Mavin et al. introduced a textual syntax for requirements, based on a precondition, an event trigger and a desired response. The syntax was intended for use in the production of Rolls-Royce aircraft engines. A review of formal specification languages aimed at requirements formalization was given in [32]. In particular, Ljungkrantz et al. [33] proposed an extended linear temporal logic ST-LTL to formally specify control logics of IEC 61131-3 programmable logic controllers in structured text. Their main improvement is in using previous variable values instead of next state operator as well as in introducing an operator for working with values of control variables at Nth step. This is the opposite of our presented approach and leads to more complicated specifications and proofs.

According to the approach presented by Kuzmin et al. [34], the value of each variable should be changed once and in only one place in the program during one iteration of the PLC cycle. Therefore, the change in value of each program variable is represented by two explicit LTL formulas:

$$\mathbf{GX}(V > \_V \implies OldValCond \vee FiringCond \vee V = NewValExpr);$$

$$\mathbf{GX}(V < \_V \implies OldValCond' \vee FiringCond' \vee V = NewValExpr'),$$

where _ is a pseudo-operator, allowing to refer to the previous state value of the variable V. This can be considered as part of our concept (see Definition 3).

Xiaohong Chen et al. [35] proposed a dynamic safety specification pattern with *Trigger* and *Postcondition* attributes that are similar to the components *trigger* and *reaction* in our pattern. However, their pattern has no direct analogs for the *final event*, *invariant condition* and *allowable delay* components. There is also a difference in time models. Time in their model is measured either in abstract real numbers (physical time) or in moments when an event occurs (logical time), while time in our model is measured either in values of the type *time* (in hours, minutes, seconds, etc.) or in the number of scan cycles.

The classic pattern system from [36] includes the most popular qualitative requirements for concurrent systems. Each pattern is described in a natural language, together with its formalization by formulas of temporal logics CTL and LTL [27], quantified regular expressions and graphical representation with GIL. In [9,37], these patterns are extended to the case of probabilistic systems and real-time systems, respectively. Some composite event patterns are suggested in [19,23]. In [38], the authors introduce patterns for quantitative characteristics of event occurrences, as well as a data pattern [39]. All mentioned approaches operate only patterns with semantics expressible in LTL and its real-time and probabilistic extensions. However, [40] shows the necessity in some cases to use the branching time logic CTL with the corresponding extensions. Recent work [17] combines descriptions of classical patterns with probabilistic and real-time patterns and provides their description in limited English. In [22], classification of patterns is presented in the form of an ontology, however the set of patterns is very limited, and they have no formal semantics. In [18], we proposed an ontology of specification patterns that combines patterns from existing requirement classifications with new patterns. This ontology can be used to express combinations of requirements of the following types: qualitative, real and branching time, with combined events, quantitative characteristics of events, and simple statements about data. Summarizing, the state-of-the-art formal systems of specification patterns seem too rich and sophisticated to express the simple needs of control software requirement engineers.

We can state that the use of requirements in the form of pure LTL formulas can lead to problems of their formalization when developing a system, therefore, our work has a novelty in the creation of an intermediate descriptive logical language that would unite all the considered approaches, and also allow describing control systems close to discussed features of control software development, with the purpose of further automatic verification.

6 Conclusion and Future Work

In this paper we have presented the Event-Driven Temporal Logic (EDTL) as the base of unambiguous and at the same time engineer-friendly specification of control software requirements. In contrast to known general-purpose specification languages, our approach offers a domain-oriented specification of discrete control software with scan cycles. Although EDTL does not use continuous time, it allows users to specify requirements for a wide class of control software.

We have proposed the EDTL-based six-component pattern to specify requirements that are independent of the internal structure of control software or the plant. We have developed two formal semantics of EDTL formulas using LTL and FOL. The constructiveness of the semantics is shown by implementing a bounded checking algorithm.

The EDTL makes description of requirements simple through the use of concepts such as inputs/outputs, falling/rising edges, events, and timeouts which are natural to the process and plant engineers. A requirements specification based on EDTL is independent of any particular verification technique.

In continuing this work, we intend to add support for pattern composition to the notation, develop consistency-checking methods for EDTL including events prioritization, formally prove the equivalence of the two proposed semantics as well as the soundness of the presented bounded-checking algorithm. We also plan to develop and implement EDTL-based verification methods for dynamic verification, model checking and deductive verification approaches and their combination.

A Bounded Checking of EDTL-requirements

In this appendix, we describe an algorithm which checks if an EDTL-requirement is satisfied for every finite initial path of a control system in some finite set of such paths. To check the EDTL-requirement tp, the algorithm follows the FOL-formula F_{tp} given in Sect. 3. For control system CS, we consider finite initial paths of length $len > 0$. The algorithm (implemented in [41]) is defined by the C-like functions `take` and `check`. The EDTL-requirements `tp` is represented by a structure with the corresponding fields `trigger`, `final` and other, the path is represented by an array `p` storing the finite history of system states, and an array `pp` stands for a set of such paths. In contrast to the bounded model checking method, this algorithm does not explore *every* initial path of a verified system.

```
bool take (struct tp, array pp) {
    for (i = 0, i < n, i++)
        if !check (tp, pp[i]) return false;
    return true;
}
bool check (struct tp, array p) {
 trig = 1;
 while (trig < len) {
  if (value(tp.trigger, p, trig, 0) {
   if (value(tp.release, p, trig, trig)) goto checked;
   fin = trig;
   while (!value(tp.final, p, fin, trig)) {
    if (value(tp.release, p, fin, trig)) goto checked;
    if (!value(tp.invariant, p, fin, trig)) return false;
    fin++;
    if (fin == len) goto checked;
   }
   del = fin;
   while (!value(tp.delay, p, del, fin) &&
          !value(tp.reaction, p, del + 1, fin)) {
    if (value(tp.release, p, del, trig)) goto checked;
    if (!value(tp.invariant, p, del, fin)) return false;
    del++;
    if (del == len) goto checked;
   }
   if (!value(tp.release, p, del, trig) &&
       value(tp.delay, p, del, fin) &&
       !value(tp.invariant, p, del, fin)) return false;
  }
  checked: trig++;
 }
 return true;
}
```

In Figure 3, we depict a class diagram based on our implementation [41] of the bounded checking algorithm for given EDTL-requirements. We implemented the EDTL-formulas as classes based on the EDTL terms. Then we encoded the R1..R5 requirements for our case study using information from Table 2. So the user can use provided classes by implementing their own system consisted of cases inherited from *CheckableReq* and overriding six methods that specify the requirements in terms of our logic. This integrates the requirements checking process into the unit testing process.

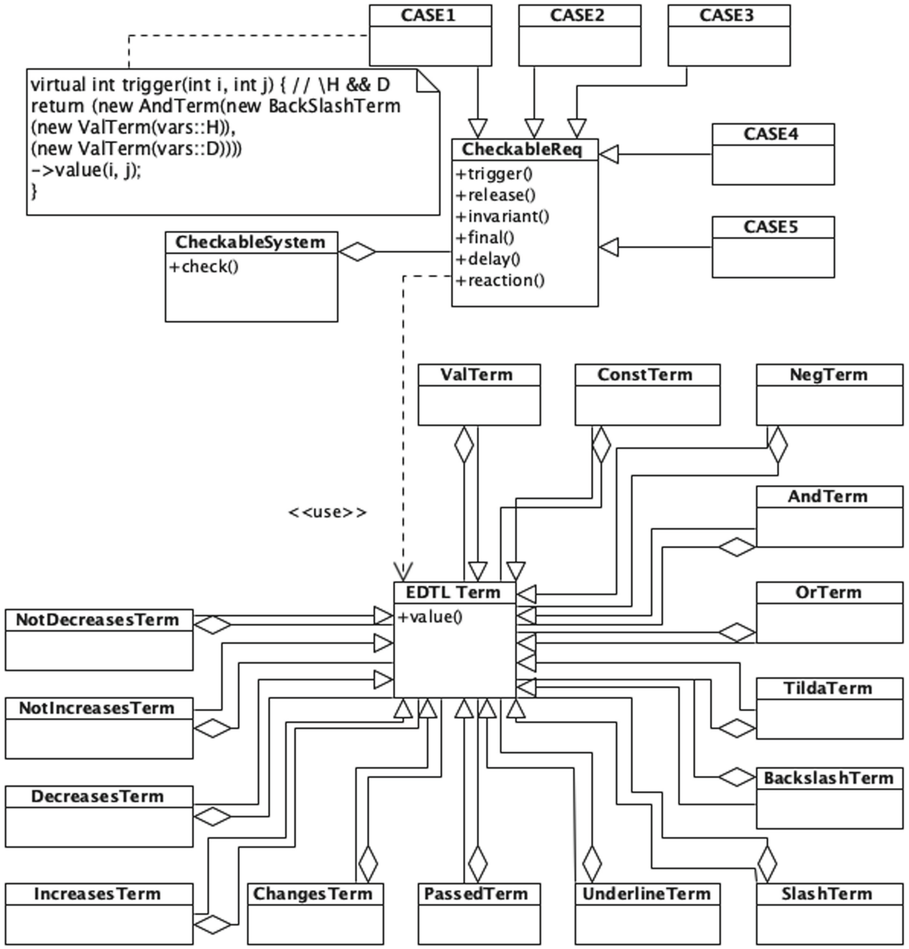

Fig. 3. Object-oriented implementation of the bounded checking algorithm for EDTL-requirements

References

1. Darvas, D., Majzik, I., Blanco Viñuela, E.: Formal verification of safety PLC based control software. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 508–522. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_32
2. Staroletov, S., et al.: Model-driven methods to design of reliable multiagent cyber-physical systems. In: Proceedings of the Conference on Modeling and Analysis of Complex Systems and Processes (MACSPro 2019), vol. 2478, pp. 74–91. CEUR Workshop Proceedings (2019)
3. Sommerville, I.: Software Engineering. Pearson Education, Harlow (2016)
4. Feng, L., et al.: Quality control scheme selection with a case of aviation equipment development. Eng. Manag. J. **32**(1), 14–25 (2020)

5. Anureev, I., Garanina, N., Liakh, T., Rozov, A., Zyubin, V., Gorlatch, S.: Two-step deductive verification of control software using reflex. In: Bjørner, N., Virbitskaite, I., Voronkov, A. (eds.) PSI 2019. LNCS, vol. 11964, pp. 50–63. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-37487-7_5
6. IEC: 61131–3 Ed. 3.0 en:2013: Programmable Controllers—Part 3: Programming Languages. International Electrotechnical Commission (2013)
7. Leveson, N., Heimdahl, M., Reese, J.: Designing specification languages for process control systems: lessons learned and steps to the future. ACM Sigsoft Softw. Eng. Notes. **24**(6), 127–145 (1999). https://doi.org/10.1145/318774.318937
8. Schneider, F., Berenbach, B.: A literature survey on international standards for systems requirements engineering. In: Proceedings of the Conference on Systems Engineering Research, vol. 16, pp. 796–805, January 2013
9. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering, pp. 372–381. ACM (2005)
10. Filipovikj, P., Nyberg, M., Rodriguez-Navas, G.: Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In: IEEE 22nd International Requirements Engineering Conference, pp. 444–450. IEEE (2014)
11. Jue, W., Song, Y., Wu, X. Dai, W.: A semi-formal requirement modeling pattern for designing industrial cyber-physical systems. In: Proceedings of IECON 2019–45th Annual Conference of the IEEE Industrial Electronics Society, Lisbon, Portugal, 2019. pp. 2883–2888 (2019)
12. Garanina, N., Anureev, I., Sidorova, E., Koznov, D., Zyubin, V., Gorlatch, S.: An ontology-based approach to support formal verification of concurrent systems. In: Sekerinski, E., et al. (eds.) FM 2019. LNCS, vol. 12232, pp. 114–130. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-54994-7_9
13. Ghosh, S., Elenius, D., Li, W., Lincoln, P., Shankar, N., Steiner, W.: ARSENAL: automatic requirements specification extraction from natural language. In: Rayadurgam, S., Tkachuk, O. (eds.) NFM 2016. LNCS, vol. 9690, pp. 41–46. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40648-0_4
14. Sarmiento, E., do Prado Leite, J.C.S., Almentero, E.: C&L: generating model based test cases from natural language requirements descriptions. In: 2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET), pp. 32–38. IEEE (2014)
15. Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements specification for process-control systems. IEEE Trans. Softw. Eng. **20**(9), 684–707 (1994). https://doi.org/10.1109/32.317428
16. Pang, C., Pakonen, A., Buzhinsky, I., Vyatkin, V.: A study on user-friendly formal specification languages for requirements formalization. In: IEEE 14th International Conference on Industrial Informatics (INDIN), pp. 676–682. IEEE (2016)
17. Autili, M., Grunske, L., Lumpe, M., Pelliccione, P., Tang, A.: Aligning qualitative, real-time, and probabilistic property specification patterns using a structured English grammar. IEEE Trans. Softw. Eng. **41**(7), 620–638 (2015)
18. Garanina, N., Zubin, V., Lyakh, T., Gorlatch, S.: An ontology of specification patterns for verification of concurrent systems. In: Proceedings of the 17th International Conference on Intelligent Software Methodology Tools, and Techniques (SoMeT_18), pp. 515–528. IOS Press, Amsterdam (2018)
19. Salamah, S., Gates, A.Q., Kreinovich, V.: Validated patterns for specification of complex LTL formulas. J. Syst. Softw. **85**(8), 1915–1929 (2012)
20. Smith, M.H., Holzmann, G.J., Etessami, K.: Events and constraints: a graphical editor for capturing logic requirements of programs. In: Proceedings of Fifth IEEE

International Symposium on Requirements Engineering, 27–31 August 2001, pp. 14–22. IEEE (2001)
21. Wong, P.Y.H., Gibbons, J.: Property specifications for workflow modelling. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 56–71. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00255-7_5
22. Yu, J., Manh, T.P., Han, J., Jin, Y., Han, Y., Wang, J.: Pattern based property specification and verification for service composition. In: Aberer, K., Peng, Z., Rundensteiner, E.A., Zhang, Y., Li, X. (eds.) WISE 2006. LNCS, vol. 4255, pp. 156–168. Springer, Heidelberg (2006). https://doi.org/10.1007/11912873_18
23. Mondragon, O., Gates, A. Q., Roach, S.: Prospec: support for elicitation and formal specification of software properties. In: Proceedings of Runtime Verification Workshop. Electronic Notes in Theoretical Computer Science, vol. 89, pp. 67–88. Elsevier (2004)
24. Garanina, N., Anureev, I., Zyubin, V., Rozov, A., Liakh, T., Gorlatch, S.: Reasoning about programmable logic controllers. Syst. Inform. **17**, 33–42 (2020)
25. Mader, A.: A classification of plc models and applications. In: Boel, R., Stremersch, G. (eds.) Discrete Event Systems. SECS, vol. 569, pp. 239–246. Springer, Boston, MA (2000). https://doi.org/10.1007/978-1-4615-4493-7_24
26. Estrada-Vargas, A.P., López-Mellado, E., Lesage, J.J.: A black-box identification method for automated discrete-event systems. IEEE Trans. Autom. Sci. Eng. **14**(3), 1321–1336 (2015)
27. Clarke, E.M., Henzinger, Th.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer International Publishing, New York (2018)
28. Zave, P.: Classification of research efforts in requirements engineering. ACM Comput. Surv. (CSUR) **29**(4), 315–321 (1997)
29. Bennaceur, A., et al.: Requirements Engineering. Handbook of Software Engineering. pp. 51–92. Springer, Cham (2019)
30. Bradner, S.: Key words for use in RFCs to indicate requirement levels (1997). http://www.ietf.org/rfc/rfc2119.txt. Accessed 17 Jan 2021
31. Mavin, A., et al.: Easy approach to requirements syntax (EARS). In: 2009 17th IEEE International Requirements Engineering Conference. IEEE (2009)
32. Pang, C., Pakonen, A., Buzhinsky, I., Vyatkin, V.: A study on user-friendly formal specification languages for requirements formalization. In: 2016 IEEE 14th International Conference on Industrial Informatics (INDIN), pp. 676–682. IEEE (2016)
33. Ljungkrantz, O., Åkesson, K., Fabian, M., Yuan, C.: A formal specification language for PLC-based control logic. In: 8th IEEE International Conference on Industrial Informatics, pp. 1067–1072. IEEE (2010)
34. Kuzmin, E.V., Ryabukhin, D.A., Sokolov, V.A.: On the expressiveness of the approach to constructing PLC-programs by LTL-specification. Autom. Control. Comput. Sci. **50**(7), 510–519 (2016). https://doi.org/10.3103/S0146411616070130
35. Chen, X., Han, L., Liu, J. Sun, H.: Using safety requirement patterns to elicit requirements for railway interlocking systems. In: 2016 IEEE 24th International Requirements Engineering Conference Workshops (REW), Beijing, 2016, pp. 296–303. IEEE (2016)
36. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, pp. 411–420. IEEE Computer Society Press (1999)
37. Grunske, L.: Specification patterns for probabilistic quality properties. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), pp. 31–40. ACM, New York, NY (2008)

38. Bianculli, D., Ghezzi, C., Pautasso, C., Senti, P.: Specification patterns from research to industry: a case study in service-based applications. In: Proceedings of 34th International Conference on Software Engineering (ICSE), pp. 968–976. IEEE (2012)
39. Halle, S., Villemaire, R., Cherkaoui, O.: Specifying and validating data-aware temporal web service properties. IEEE Trans. Softw. Eng. **35**(5), 669–683 (2009)
40. Post, A., Menzel, I., Podelski, A.: Applying restricted English grammar on automotive requirements—does it work? A case study. In: Berry, D., Franch, X. (eds.) REFSQ 2011. LNCS, vol. 6606, pp. 166–180. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19858-8_17
41. Staroletov, S.: EDTL: Object-oriented implementation of the bounded checking algorithm for EDTL-requirements (2020). https://doi.org/10.5281/zenodo.4445663. Accessed 17 Jan 2021

Author Queries

Chapter 7

Query Refs.	Details Required	Author's response
AQ1	This is to inform you that corresponding author has been identified as per the information available in the Copyright form.	
AQ2	Please note that as per Springer guideline private email address of the author will not be displayed.	