Вычислительные технологии, 2022, том 27, № 1, с. 88–100. © ФИЦ ИВТ, 2022 ISSN 1560-7534
Computational Technologies, 2022, vol. 27, no. 1, pp. 88–100. © FRC ICT, 2022 eISSN 2313-691X



DOI:10.25743/ICT.2022.27.1.007

# Towards unit testing of event-driven control requirements

Staroletov S. M.*, Anureev I. S.
Institute of Automation and Electrometry, 630090, Novosibirsk, Russia
*Corresponding author: Staroletov Sergey M., e-mail: serg_soft@mail.ru
*Received August 19, 2021, revised October 26, 2021, accepted November 12, 2021.*

Testing is a generally accepted method to control software quality, although it is not completely reliable. Nevertheless, this method integrates extremely well into development environments and continuous integration practices. In this paper, we briefly review the behavioral patterns that we have previously developed for the logical description of the programmable logic controllers (PLC) operations using tabular properties. We also present a diagram for the checking algorithm of a bounded model to investigate the feasibility of such properties. We describe how to implement the terms and formulas that provide the behavior patterns of PLC programs in an object-oriented programming language (C++ in this case). After the black box assessment for the values of the control variables for inputs and outputs of the system has been set, we show how convenient it is to describe the requirements in the form of our instantiated classes. This description allows integrating the unit testing process for the checking requirements of the PLC programs.

*Keywords*: requirements engineering, unit testing, control software, PLC.

*Citation*: Staroletov S.M., Anureev I.S. Towards unit testing of event-driven control requirements. Computational Technologies. 2022; 27(1):88–100. DOI:10.25743/ICT.2022.27.1.007.

## Introduction

While the first standard for unit testing was proposed by ANSI/IEEE in 1987 [1], the start of wide industrial applicability of this method and its formalization into a methodology was done in 1998 by Kent Beck and Erich Gamma with their famous JUnit implementation [2]. Since then, testing at the code level is performed by the developers themselves, by implementing tests that aim to cover all the methods and lines of code in them. If we take a formal look at the unit testing process [3], then the result of such testing is checking the correctness of the expression:

$$\bigwedge_{M_i \in Testcase} M_i(x_1..x_n) == Ret_{expect}(M_i)$$

where $M_i(x_1..x_n)$ is the result of the tested method (function) with the specified parameters, $Ret_{expect}$ is the expected return value of the method (or its effect on the state of the class/ module). Methods (functions) are grouped into test cases. Conjunction means that if one of the tested values does not correspond to the expected, the operation of the entire test suite is considered incorrect.

The advantages of the methodology are:
- standardization of the testing process at the code level;
- support by the developer community;
- implementation for all modern programming languages;
- support from integrated development environments (IDEs) [4];
- ability to use in continuous integration (CI) processes;
- conducting regression testing to check that the new code still passes pre-written tests;
- ability to change the sequence of test development and writing, test-driven development (TDD) [5].

At the same time, the methodology has one significant drawback associated with the fact that the developer, when writing tests, continues to think at the level of the code, and not at the technical specifications for the development of the system.

In the construction of control systems for technological processes, such a question is especially relevant, since here the development is usually carried out not according to the personal ideas of the programmer, but according to the rigorously detailed technical specifications for the system operation.

The long-term goal of our research group is to develop logical formalisms for describing the behaviour of programs for programmable logic controllers (PLCs) [6]. In other words, we would like to work out some requirements engineering [7] practices for such devices. By PLC we mean a microcontroller of higher software and hardware abstraction. This scientific direction arose among the group by analyzing various implemented software control systems for technological processes. After implementation and verification of several such systems [8, 9], we began to think about the task to identify the main logical approaches and methods of formalizing (1) the low-level requirements based on the analysis of control cycles [10], and (2) high-level requirements based on the analysis of the requirements for the functionality of the control program. We are concentrating on requirements which come from technical documentation and industrial specifications.

Inspired by the approach of software design patterns [11], we set out the goal to develop some basic patterns for PLC programs in terms of their logic of functioning. According to our hypothesis, modelling of such programs is possible using tabular properties (i. e., using so called rule-based systems [12]). Each row in the table represents a requirement, while the columns in the table describe parts of the overall behavioural pattern. In each cell of the table, there can be a formula of logic, specially oriented to the evaluation of signals. If it is possible to set the semantics of the satisfiability of such patterns and develop an algorithm for checking them, then it is possible to accomplish the process of checking tabular properties within the process of running tests.

The main contribution of this work is to apply the object-oriented programming approach for PLC requirements based on our logical formalism, in order to start checking the compliance of the system with these requirements within the unit testing process. Further, in Section 1, we give some necessary information about logic patterns for control systems; in Section 2 we discuss class diagrams for logical terms and the system under test; while in Section 3, we consider the implementation of all this.

## 1. On event-driven temporal logic patterns

In this section, we recall the slightly modified syntax and semantics of the proposed *Event-driven Temporal Logic* (EDTL) notation for requirements from our FSEN'21 paper [13]. The

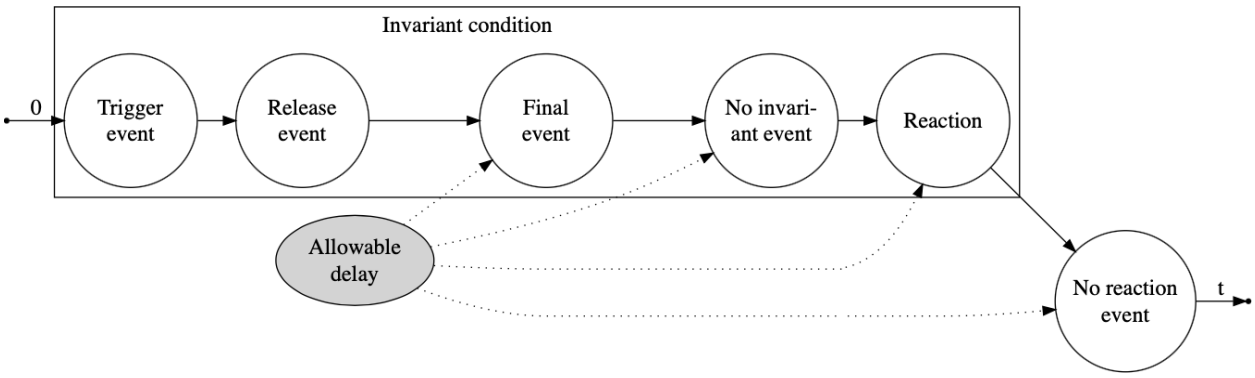


Fig. 1. A concept of specification for event-driven control requirements [13]

modifications mainly consist in improved notation, and everything necessary is presented in the current paper without the need to once again refer to the previously cited work.

### 1.1. EDTL-requirements

An EDTL requirement $\mathcal{R}$ for a PLC program is a sextuple:

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

The six-component requirement can be explained using the following informal description in natural language:

> Following each **trigger** event, the **invariant** must hold until an occurrence of either **release** event or **final** event. The **invariant** must also be hold 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.

Our graphical intuition for the temporal organization of EDTL-attributes is shown in Fig. 1.

### 1.2. EDTL-formulas

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

So, EDTL-formulas are constructed from Boolean terms by standard Boolean operations as well as our special operations for expressing instant control system changes as we show further. If $\phi$ and $\psi$ are EDTL-formulas then:

- ETDL-term of type *bool* is an atomic EDTL-formula;
- $\phi \wedge \psi$ is the conjunction of $\phi$ and $\psi$;
- $\phi \vee \psi$ is the disjunction of $\phi$ and $\psi$;
- $\neg\phi$ is the negation of $\phi$;
- $\backslash\phi$ is the falling edge: the value of $\phi$ changes from $true$ to $false$;

- $/\phi$ is the rising edge: the value of $\phi$ changes from $false$ to $true$;
- $\_\phi$ is low steady-state: the value of $\phi$ remains equal to $false$;
- $\sim \phi$ is high steady-state: the value of $\phi$ remains equal to $true$.

As we can see, such formulas are closely related to the requirements in natural language arising from the technical specifications for the development of the system.

### 1.3. Semantics of EDTL-terms

The function $\mathcal{V}$ defines semantics (value) of EDTL-terms at the time point $i$ on the path $\pi$ (a sequence of states, where $\pi(k)$ is a state at the time point $k$) with the timer point $j$ (a time point on a path to define the moment of starting a timer):

- if $c$ is a constant, then $\mathcal{V}(c, \pi, i, j) = c$;
- if $x$ is a variable, then $\mathcal{V}(x, \pi, i, j) = acc(x, \pi(i))$;
- if $u$ is a time term, then $\mathcal{V}(u, \pi, i, j) = time(u, \pi(i))$;
- $\mathcal{V}(f(u_1, \ldots, u_n), \pi, i, j) = intr(f)(\mathcal{V}(u_1, \pi, i, j), \ldots, \mathcal{V}(u_n, \pi, i, j))$;
- $\mathcal{V}((u), \pi, i, j) = \mathcal{V}(u, \pi, i, j)$.

  Let $u$ be not a term of type $time$ and $i > 0$:
- $\mathcal{V}(changes(u), \pi, i, j) = true \Leftrightarrow \mathcal{V}(u, \pi, i-1, j) \neq \mathcal{V}(u, \pi, i, j)$;
- $\mathcal{V}(increases(u), \pi, i, j) = true \Leftrightarrow \mathcal{V}(u, \pi, i-1, j) < \mathcal{V}(u, \pi, i, j)$;
- $\mathcal{V}(decreases(u), \pi, i, j) = true \Leftrightarrow \mathcal{V}(u, \pi, i-1, j) > \mathcal{V}(u, \pi, i, j)$.

  Let $u$ be a term of type $time$ and $i > 0$:
- $\mathcal{V}(passed(u), \pi, i, j) = true \Leftrightarrow i \geq j + \mathcal{V}(u, \pi, i, j)$, i.e. $\mathcal{V}(u, \pi, i, j)$ time steps have passed after the timer point $j$.

Here $acc(x, \pi(i))$ is a value of variable $x$ in state $\pi(i)$, $time(u, \pi(i))$ returns the number of scan cycles which will be passed during time $u$ with the time point $i$ for the path $\pi$, $intr(f)$ is an interpretation of function $f$ [14].

### 1.4. On bounded checking the EDTL-requirements

In Fig. 2, we show a block diagram of an algorithm that 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 $\mathcal{R}$, the algorithm follows the constructive way based on FOL-formula $F_{tp}$ given in our paper [13] (and our natural-language intuition described in Section 1.2). We use the $\mathcal{V}$-function for EDTL-terms presented above. Here we consider finite initial paths of length $len > 0$.

## 2. Object-oriented modelling

In Fig. 3, we demonstrate a UML-modelling [15] result for EDTL-terms (and EDTL-formulas as a special case of EDTL-terms of type *bool*), described in Section 1. We have divided the original figure from an appendix of [13] to describe it more clearly and introduce the new *Havoc* class which we will explain later. We model the abstract class *Term*, and then create its descendants according to the definition of EDTL formula in Section 1.2. To provide the realization of semantics for the behaviour of each term, we have to define a virtual method *value*, which is overridden in the descendants and should actually implement the $\mathcal{V}$-function of the term behaviour according to Section 1.3.

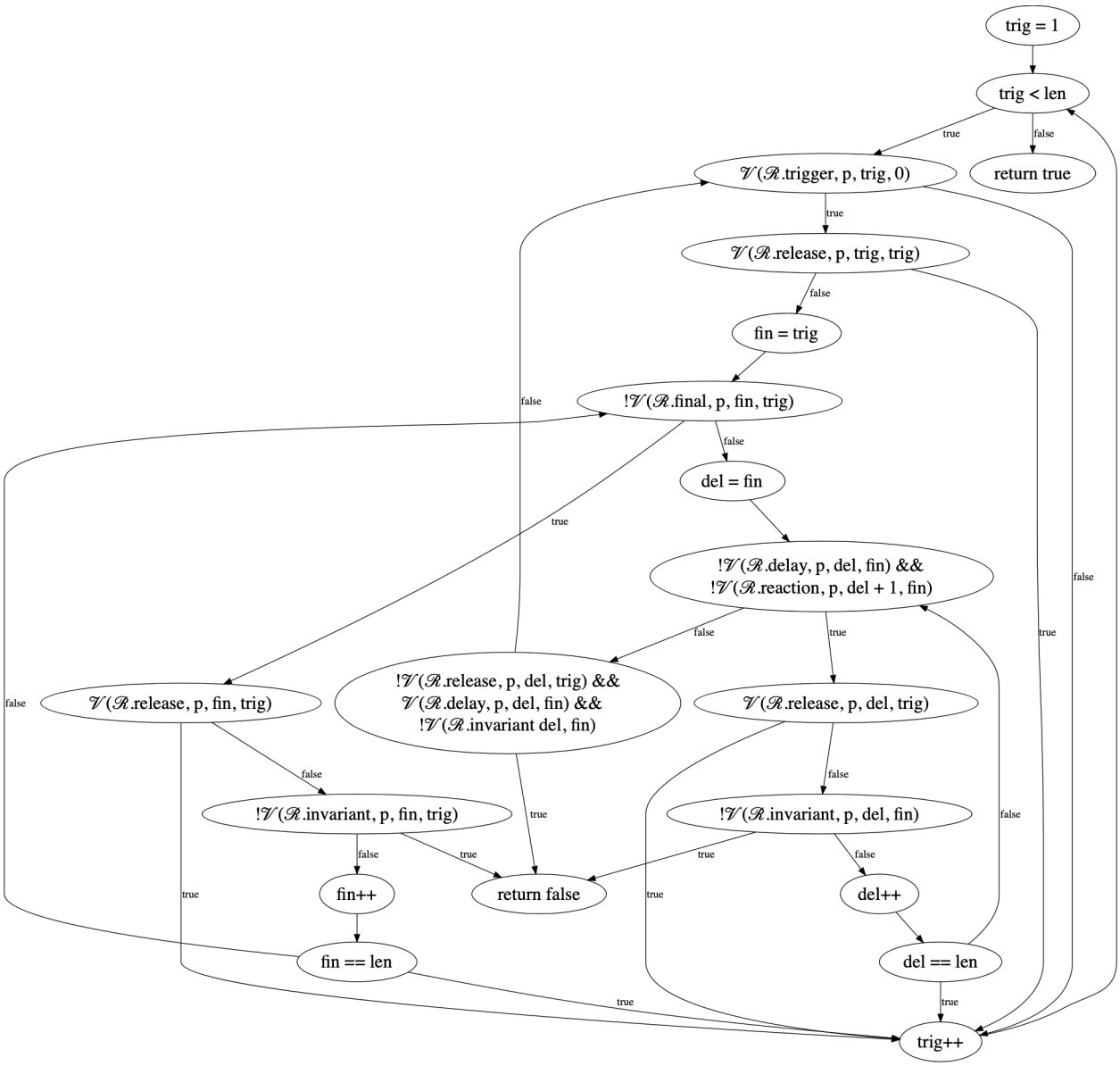


Fig. 2. A scheme for our algorithm to check the feasibility of an EDTL-requirement

To model a unit-testable system (Fig. 4), we define the *CheckableSystem* class with a method to check it using the algorithm depicted in Fig. 2. Requirements are specified as a list of *CheckableReq* subclasses, which must have methods overridden for all six components of the pattern. In the fugure, we drew five cases (CASE1-CASE5), which means that there are five different requirements for a particular system as six-component tuples, and the user should provide implementations for *trigger()*, *release()*, *invariant()*, *final()*, *delay()* and *reaction()* methods. Each such method defines an EDTL-formula that is built from these terms.

For the process of checking such requirements, it is necessary to specify a sequence of input and output signals in the form of Boolean vectors. Here we must abstract from the implementation of the system because we only need the values of its control variables according to the black-boxing principle [16]. Such values are operated by the *Havoc* singleton

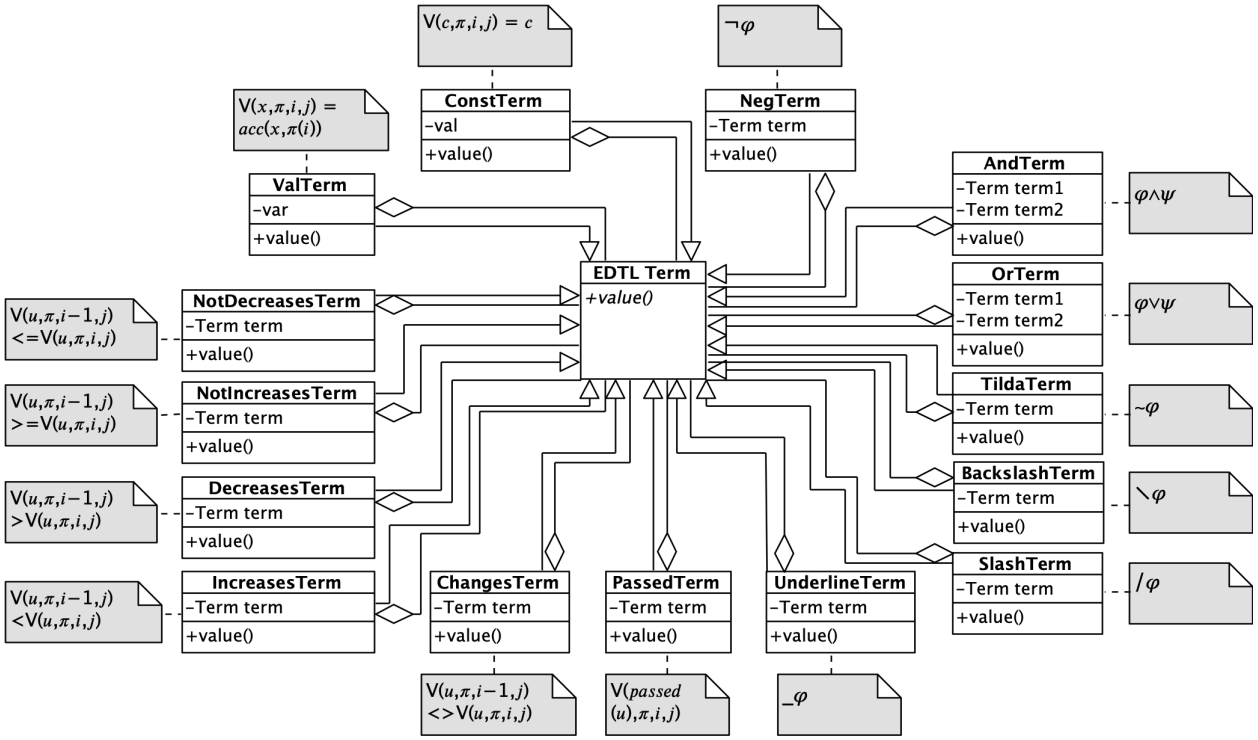


Fig. 3. An EDTL term

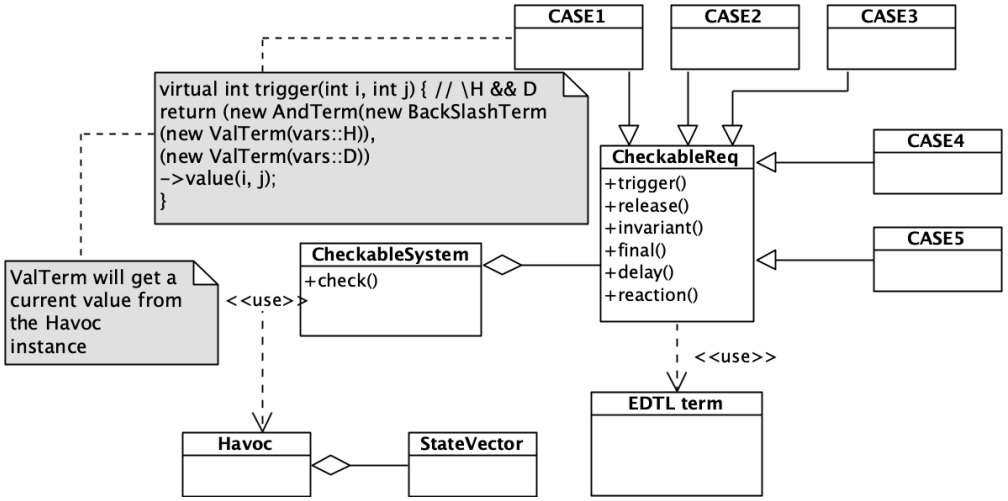


Fig. 4. A model of system with sets of requirements in the form of EDTL-terms

class (its name corresponds to a Microsoft Research tool [17]), which provides the current vector of their values to the *ValTerm* term, where such values are required at the current moment when its *value* method is called.

Let us consider the process of checking requirements in the form of a sequence diagram (Fig. 5). First, the user sets vectors of sequences of variable values on which the requirements will be checked. The vector data is put into the *Havoc* class (we make this class responsible for providing the current values of the variables). Next, the *check()* method of the system is called, which returns a Boolean sign of its correctness or incorrectness (or, as for control

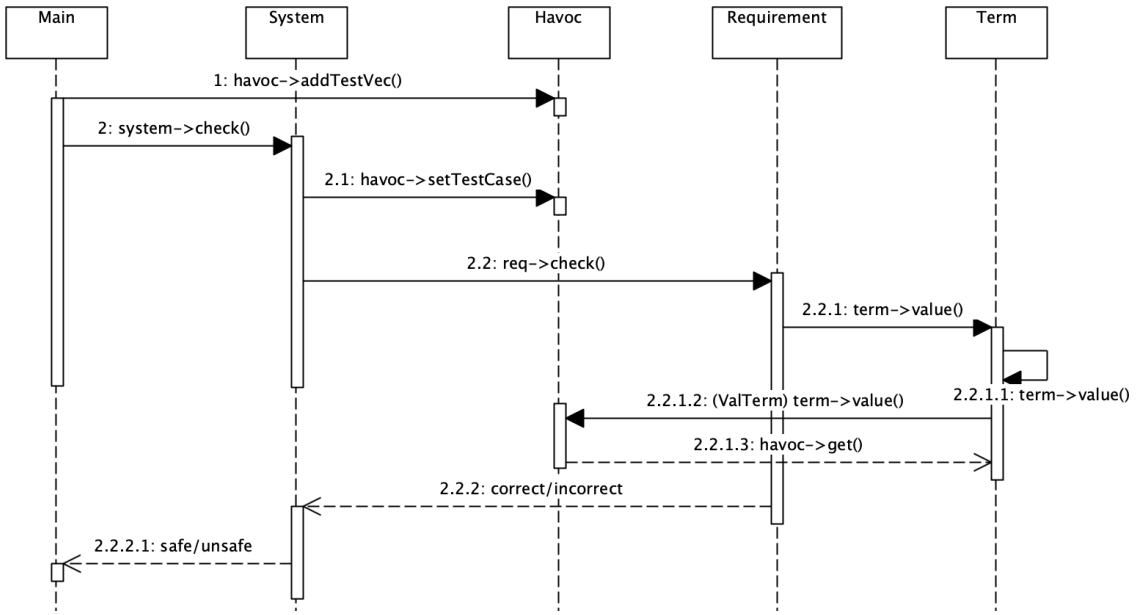


Fig. 5. Requirements checking process in action

systems, safe/unsafe). Inside it, all possible test vectors are requested from *Havoc*, and on each test vector for each requirement the bounded checking algorithm (shown in Fig. 2) is launched. Each requirement, as we have already discussed, contains six components, expressed as EDTL-terms. All of them are required for our checking algorithm to operate, so recurred calls are made to the predefined *value()* method to get the values of the $\mathcal{V}$-function. When asking for a *ValTerm* value, *Havoc* provides the current value of the requested variable from the current test vector.

## 3. Object-oriented implementation

In this section, we show code snippets to describe main ideas of the discussed concepts implementation as well as the usage of them to provide a system description in the form of event-oriented structured requirements expressed in EDTL-terms.

So, we are going to discuss:

- Persistent part, in the form of abstract and reusable classes for terms as well as requirements and the composite system with the bounded checking algorithm.
- User's part, that shows how to create the above classes, define and verify specific requirements.

### 3.1. The implementation of EDTL-terms

As an example of term implementation, we consider *BackSlashTerm*, which corresponds to the falling edge of the situation as defined in Section 1. It is derived from *Term*, and overrides the virtual method *value*:

```
class BackSlashTerm : public Term {
 private:
  //a root term
  Term *term;
```

```
 public:
  //constructor to obtain a root term
  BackSlashTerm(Term *term) : term(term) {}
  //v-function redefinition
  int value(int i, int j) {
    return (i != 0) && !term->value(i - 1, j) &&
    term->value(i, j);
  }
};
```

Other terms are implemented in the same manner in accordance with the definitions from Section 1.3, Section 1.4, and the overall plan in Fig. 3.

### 3.2. Instantiating test cases for the system

Next, we have to instantiate the system for testing and set the list of requirements for it as subclasses of *CheckableReq*:

```
CheckableSystem *system = new CheckableSystem();
system->addReqs({
 new CASE1(), new CASE2(), new CASE3(), new CASE4(), new CASE5()
});
```

Note, we use *std::initializer_list* [18] to pass the list of parameters in more functional and easy-readable way.

After the actual elaboration and specification of the system requirements (in this example, instantiating the CASE1-CASE5 classes for five requirements), as well as setting test vectors of control variable values (we will consider this in detail below), the system is checked with one high-level call:

```
if (system->check())
    cout << "System is safe" << endl;
else
    cout << "System is unsafe " << endl;
```

### 3.3. How to implement user's code for the system using our framework

Referring to the slightly modified example from the article [13], we consider a simple sanitizer device (Fig. 6) to set its requirements from the code. This system reacts to the appearance of hands (control variable $H$) by turning on the disinfector (variable $D$) for some time interval.

For one of the cases, the requirement *"If the disinfector is on, it turns off after 2 seconds without hands"* was elaborated. Using $H$ and $D$ variables, this requirement can be expressed as a tuple:

$$\mathcal{R}_{case1} = (\textbf{trigger}=\backslash H \&\& D, \textbf{invariant}=D, \\ \textbf{final}=passed(2), \textbf{delay}=true, \textbf{reaction}=!D, \textbf{release}=H).$$

All six components of the pattern for this case should be defined in the code by using polymorphic methods:

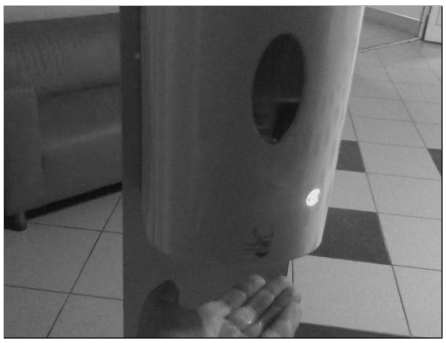

Fig. 6. Simple anti-Covid-19 sanitizer as a demo system to specify requirements

```
class CASE1 : public CheckableReq {
  //in this subclass, we have to define
  //realizations for 6 methods
  virtual int trigger(int i, int j) {
    // returns an OOP representation
    // for \H && D formula
    return (new AndTerm(
      new BackSlashTerm(new ValTerm(Vars::H)),
      new ValTerm(vars::D))
    )->value(i, j);
  }
  virtual int release(int i, int j) {
    // returns an OOP representation
    // for H term
    return (new ValTerm(Vars::H))->value(i, j);
  }
  virtual int final(int i, int j) {
    // returns an OOP representation for
    // ``passed 2s'' term
    return (new PassedTerm(new ConstTerm(2)))->value(i, j);
  }
  virtual int delay(int i __unused, int j __unused) {
    //since this part is not used, we return true
    return true;
  }
  virtual int invariant(int i, int j) {
    // returns an OOP representation for
    // D term
    return (new ValTerm(Vars::D))->value(i, j);
  }
  virtual int reaction(int i, int j) {
    // returns an OOP representation for
    // !D formula
    return (new NegTerm(new ValTerm(Vars::D)))->value(i, j);
  }
};
```

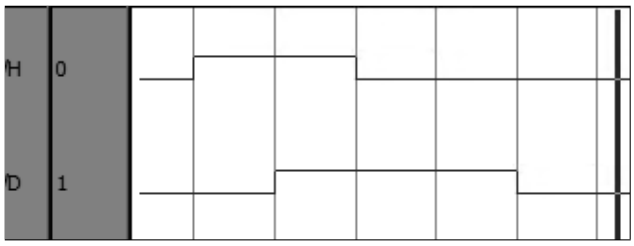


Fig. 7. Waveform representation of variables values over time

To ensure further testing, it is necessary to add test vectors of values of the system control variables against which we are going to check the requirements. Recall that a typical unit test has the system under test (or its "stub" or "fake", see [19]), expected values, and real values. According to our goals, the system is represented by a black box and replaced by the values of the control variables over time. The algorithm for checking the requirement (Fig. 2) on the basis of a set of six-component requirements and passed values of variables returns whether such a model of the system meets the specified requirements or not.

So, we need to pass a set of input and output vectors to the *Havoc* class:

```
havoc->addTestVector({
    TestVec{{0, 1, 1, 0, 0, 0}, Vars::H},
    TestVec{{0, 0, 1, 1, 1, 0}, Vars::D}
}); //good
havoc->addTestVector({
    TestVec{{1, 1, 0, 0, 0, 0}, Vars::H},
    TestVec{{1, 1, 0, 0, 1, 1}, Vars::D}
}); //violates some requirements
```

We have also to validate whether the size of passed vectors in a call is the same and all variables of the system are specified. In Fig. 7, we show the graphical representation of input test vector in the first *addTestVector()* call. According to this, hands appear first, after which the disinfector turns on. After removing the hands it continues to work for some time interval (two time steps in this case). Such test vectors can be obtained from real operating devices using digital logic analyzers [20]. Note, if we set requirements in units of time, such as seconds or milliseconds, we need to convert this time into timesteps by multiplying by some constant, for simplicity in this example it is equal to 1.

In this way, we have showed how to turn the requirements validation into tests.

Although we are still on the initial path in creating test cases for this formalism, some patterns for more complex control systems have already been discussed in a CSMML workshop publication [21].

## 4. Related work

With regard to modelling the behaviour of PLCs, in this area there are known works with the use of automata [22]. Recently, temporal logics have become actively used [23, 24]. Some examples of dynamic specifications of security properties are given in the work [25], but they, as a rule, do not describe all six components, as we have done. Some issues related to formulate properties related to clocks in a temporal logic are discussed in [26]. The paper [27]

describes IEEE standard for a logical language, in which it is proposed to combine temporal operators and regular expressions to specify the behaviour of signals.

As to the industrial application of the requirements testing methodology in the development process, we can note here the behaviour-driven development (BDD) [28] methodology. Currently, this is essentially the only specification approach that is used in software engineering and supported by many development tools [29]. At the same time, in a development environment, specifications are launched in the tests window, we can see which specification has passed and which has not, which forces developers to think within the framework of the specifications [3]. We also strive for such a toolkit.

## Conclusion and future work

While many research groups are developing formal verification methods to prove the correct operation of various software systems with respect to their formal models, the industrial applicability of such developments remains extremely low. In this work, we have developed a logic-to-program transition: a framework for checking the behaviour of PLC programs by unit testing using a logical formalism in the form of implemented EDTL-terms, and a custom model-checking algorithm inside. We have implemented a demo system in C++ which is available on GitHub [30].

In terms of further work, we are thinking about implementing support for such solutions from a development environment and setting requirements as annotations in PLC-oriented domain-specific languages [31].

**Acknowledgements.** 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.

## References

[1] ANSI/IEEE. IEEE standard for software unit testing. ANSI/IEEE Std 1008-1987. N.Y.: IEEE Computer Society; 1987: 28. DOI:10.1109/IEEESTD.1986.81001.

[2] **Beck K., Gamma E.** Test infected: Programmers love writing tests. Java Report. 1998; 3(7):37–50.

[3] **Staroletov S.** Basics of software testing and verification. Saint Petersburg: Lanbook Publishing; 2018. Available at: `https://e.lanbook.com/book/138181`.

[4] Jetbrains. IDEA. Testing, 2021. Available at: `https://www.jetbrains.com/help/idea/testing.html` (accessed January 25, 2022).

[5] **Beck K.** Test-driven development: By example. Boston: Addison-Wesley Professional; 2003: 220.

[6] **Hanssen D.H.** Programmable logic controllers: A practical approach to IEC 61131-3 using CODESYS. Wiley Online Library; 2015: 416.

[7] **Zave P.** Classification of research efforts in requirements engineering. ACM Computing Surveys (CSUR). 1997; 29(4):315–321. DOI:10.1145/267580.267581.

[8] **Liakh T.V., Garanina N.O., Anureev I.S., Zyubin V.E.** Verifying Reflex-software with SPIN: Hand dryer case study. 2020 21st International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM). Chemal: IEEE; 2020: 210–214. DOI:10.1109/EDM49804.2020.9153545.

[9] **Staroletov S.** Developing automata-based control software for water purification and normalization. 2021 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM). Sochi: IEEE; 2021: 541–546. DOI:10.1109/ICIEAM51226.2021.9446349.

[10] **Anureev I.S., Zyubin V.E., Staroletov S.M., Liakh T.V., Rozov A.S., Gorlatch S.P.** Temporal logic for programmable logic controllers. Modelling and Analysis of Information Systems. 2020; 27(4):412–427. DOI:10.18255/1818-1015-2020-4-412-427.

[11] **Gamma E., Helm R., Johnson R., Vlissides J.** Design patterns. Elements of reusable object-oriented software. Massachusetts: Addison-Wesley Publishing; 1995: 383.

[12] **Ligeza A.** Toward logical analysis of tabular rule-based systems. International Journal of Intelligent Systems. 2001; 16(3):333–360.

[13] **Zyubin V., Anureev I., Garanina N., Staroletov S., Rozov A., Liakh T.** Event-driven temporal logic pattern for control software requirements specification. Fundamentals of Software Engineering. FSEN 2021. Lecture Notes in Computer Science. 2021; 12818:92–107. DOI:10.1007/978-3-030-89247-0_7.

[14] **Cousot P., Cousot R.** Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Los Angeles: ACM; 1977: 238–252. DOI:10.1145/512950.512973.

[15] **Miles R., Hamilton K.** Learning UML 2.0: A pragmatic introduction to UML. Sebastopol: O'Reilly Media, Inc.; 2006: 290.

[16] **Beizer B.** Black-box testing: Techniques for functional testing of software and systems. Chichester: Wiley; 2008: 320.

[17] **Ball T., Hackett B., Lahiri S.K., Qadeer S., Vanegue J.** Towards scalable modular checking of user-defined properties. International Conference on Verified Software: Theories, Tools, and Experiments. Edinburgh: Springer; 2010: 1–24. DOI:10.1007/978-3-642-15057-9_1.

[18] cppreference.com. std::initializer list, 2011. Available at: `https://en.cppreference.com/w/cpp/utility/initializer_list` (accessed January 25, 2022).

[19] Microsoft. Isolate code under test with Microsoft Fakes, 2020. Available at: `https://docs.microsoft.com/en-us/visualstudio/test/isolating-code-under-test-with-microsoft-fakes?view=vs-2019` (accessed January 25, 2022).

[20] Kingst Electronics. Virtual instruments user guide (v3.5), 2020. Available at: `http://res.kingst.site/kfs/doc/Kingst_Virtual_Instruments_User_Guide.pdf` (accessed January 25, 2022).

[21] **Garanina N., Koznov D.** Static checking consistency of temporal requirements for control software. International Conference on Model and Data Engineering. CSMML Workshop. Tallinn: Springer; 2021: 189–203. DOI:10.1007/978-3-030-87657-9_15.

[22] **Dierks H.** PLC-automata: A new class of implementable real-time automata. International AMAST Workshop on Aspects of Real-Time Systems and Concurrent and Distributed Software. Palma: Springer; 1997: 111–125. DOI:10.1007/3-540-63010-4_8.

[23] **Ljungkrantz O., Akesson K., Fabian M., Ebrahimi A.H.** An empirical study of control logic specifications for programmable logic controllers. Empirical Software Engineering. 2014; 19(3):655–677. DOI:10.1007/s10664-012-9232-x.

[24] **Kuzmin E., Ryabukhin D., Sokolov V.A.** On the expressiveness of the approach to constructing PLC-programs by LTL-specification. Automatic Control and Computer Sciences. 2016; 50(7):510–519. DOI:10.3103/S0146411616070130.

[25] **Chen X., Han L., Liu J., Sun H.** Using safety requirement patterns to elicit requirements for railway interlocking systems. 24th International Requirements Engineering Conference Workshops (REW). Beijing: IEEE; 2016: 296–303. DOI:10.1109/REW.2016.055.

[26] **Eisner C., Fisman D., Havlicek J., McIsaac A., Van Campenhout D.** The definition of a temporal clock operator. International Colloquium on Automata, Languages, and Programming. Eindhoven: Springer; 2003: 857–870. DOI:10.1007/3-540-45061-0_67.

[27] 1850–2005 — IEEE standard for property specification language (PSL). DOI:10.1109/IEEESTD.2005.97780. Available at: `https://ieeexplore.ieee.org/document/1524461/citations#citations`.

[28] **Wynne M., Hellesoy A., Tooke S.** The Cucumber book: Behaviour-driven development for testers and developers. Pragmatic bookshelf. Boston: Addison-Wesley; 2017: 336.

[29] **Smart J.F.** BDD in action. N.Y.: Manning Publications; 2014: 353.

[30] **Staroletov S.** EDTL implementation, 2021. Available at: `https://github.com/SergeyStaroletov/EDTL` (accessed January 25, 2022).

[31] **Zyubin V.E., Liakh T.V., Rozov A.S.** Reflex language: A practical notation for cyber-physical systems. System Informatics. 2018; 12:85–104. DOI:10.31144/si.2307-6410.2018.n12.p85-104 Available at: `https://system-informatics.ru/files/article/lyahrozovzubin2018_0.pdf` (accessed January 25, 2022).

Вычислительные технологии, 2020, том 25, № 6, с. 88–100. © ФИЦ ИВТ, 2020 ISSN 1560-7534
Computational Technologies, 2020, vol. 25, no. 6, pp. 88–100. © FRC ICT, 2020 eISSN 2313-691X

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

DOI:10.25743/ICT.2022.27.1.007

## На пути к модульному тестированию событийно-управляемых требований

С. М. Старолетов*, И. С. Ануреев

Институт автоматики и электрометрии СО РАН, 630090, Новосибирск, Россия

* Контактный автор: Старолетов Сергей Михайлович, e-mail: `serg_soft@mail.ru`

*Поступила 19 августа 2021 г., доработана 26 октября 2021 г., принята в печать 12 ноября 2021 г.*

### Аннотация

Тестирование — общепринятый метод контроля качества программного обеспечения, хотя о полной надежности программ при таком подходе говорить не приходится. Тем не менее этот метод очень хорошо интегрируется в среды разработки и применим при непрерывной интеграции. В статье кратко рассмотрены шаблоны поведения, которые ранее разработаны для логического описания операций программируемых логических контроллеров (ПЛК) с использованием табличных свойств. Представлена схема ограниченного алгоритма проверки модели для контроля выполнимости этих свойств. Описано, как реализовать термы и формулы, составляющие модели поведения программ ПЛК, на объектно-ориентированном языке программирования. После того как была проведена абстракция значений входов и выходов управляющих переменных системы на основе подхода “черного ящика”, показано, насколько удобно описывать требования в форме наших экземпляров классов. Это описание позволяет интегрировать процесс проверки требований ПЛК-программ в процесс модульного тестирования.

*Ключевые слова*: инженерия требований, модульное тестирование, управляющее программное обеспечение, программируемый логический контроллер.

*Цитирование*: Staroletov S.M., Anureev I.S. Towards unit testing of event-driven control requirements. Computational Technologies. 2022; 27(1):88–100. DOI:10.25743/ICT.2022.27.1.007.

**Благодарности.** Работа профинансирована из государственного бюджета Российской Федерации (проект ИАиЭ № АААА-А19-119120290056-0).