Static Checking Consistency of Temporal Requirements for Control Software

Natalia Garanina[1] and Dmitry Koznov[2]

[1] A.P. Ershov Institute of Informatics Systems,
Institute of Automation and Electrometry, Novosibirsk, Russia
garanina@iis.nsk.su
[2] St. Petersburg University, Russia
d.koznov@spbu.ru

Abstract. In this paper, we propose an approach to checking consistency of control software requirements described using pattern-based notation. This approach can be used at the beginning of control software verification to effectively identify contradicting and incompatible requirements. In this framework, we use pattern-based Event-Driven Temporal Logic (EDTL) to formalize the requirements. A set of requirements is represented as a set of EDTL-patterns which formal semantics defined by formulas of linear-time temporal logic LTL. Based on this semantics, we define the notion of requirement inconsistency and describe restrictions on the values of pattern attributes which make requirements inconsistent. Checking algorithm takes as an input pairs of requirements and compare their attributes. Its output is sets of consistent, inconsistent and incomparable requirements.

Keywords: Requirement engineering · Requirement consistency · Formal semantics · Linear Temporal Logic

1 Introduction

The long-term goal of our work is formal verification of control software (CS), specified in the process-oriented paradigm, in particular, written using the Domain-Specific Language (DSL) Reflex and poST [1,3,9].

In [16], for control software requirements, we propose a pattern-based specification formalism EDTL (Event-Driven Temporal Logic). EDTL-specifications has the following features: (1) a pattern form includes CS-specific concepts (such as trigger, reaction, etc.) which allow engineers to easily define requirements; (2) formal semantics provides possibility to apply formal verification methods. In particular, we can use model checking methods if EDTL-semantics is LTL-formulas [4].

Pattern-based systems for the development of requirements and their formal verification are an active topic of research for a long time [2,10,11,13,14,15]. Patterns are parameterized natural language expressions that describe typical requirements for a system behavior. Typically, parameters of patterns are system

events or their combinations (for example, in the pattern "Event *Device_ enabled* will occur", *Device_ enabled* is a parameter). Patterns usually have strict formal semantics. Patterns make it easy for requirement engineers to specify and verify typical system requirements.

Usually, control software systems have to meet a large number of requirements. Therefore, before applying formal verification methods, it is reasonable to check this list for consistency of the requirements in order to avoid unnecessary use of expensive formal methods. In this paper, we propose a method for checking the consistency of a requirements set based on pairwise comparing requirements with LTL-semantics. This pairwise consistency checking is propositional in the sense that it uses only the pattern representation of requirements and does not use a description of the control system. While this method only gives an exact answer for requirements that satisfy certain constraints, it can significantly reduce the time and cost of full consistency checking a set of requirements.

Checking consistency of requirements is rather necessary process in concurrent system development. A semi-automatic method presented in [12] helps to write use-case scenarios in a natural language iteratively and to verify temporal consistency of the behavior encoded in them. This method is used to check consistency of the textual specification of the use-case scenarios taking into account specific annotations written by user. These annotations are converted into temporal logic formulas and verified within the framework of a formal behavior model. A commercial requirement engineering tool for embedded systems Argosim [17] allows software engineers to test the systems and perform inconsistency checking for requirements. Ontology based checking is suggested in [7].

The rest of the paper is orgaised as follows. In Section 2, we give a brief description of syntax and semantics for EDTL-requirements. Section 3 describes the method of checking inconsistency for two requirements which patterns have non-constant attributes only. Section 4 uses results of the previous section to define a function that verify inconsistency of two requirements. In Section 5, we describe the algorithms that implement our method. Section 6 is conclusion.

2 Event-Driven Temporal Logic Pattern

Let us define EDTL-requirements. Informally, a EDTL-requirement is a combination of system events bounded by specified temporal interrelations. These events are attributes of the EDTL-requirement.

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

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

We give the informal meaning of the attributes and interrelations:

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.

The following natural language description of EDTL-requirement semantics corresponds to this informal description:

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.

The values of EDTL-attributes are EDTL-formulas. The EDTL-formulas are divided into two classes: state formulas and event formulas. Informally, the state formulas assert about system variables' values in a given time moment, but the event formulas assert about events that just happen or not happen, i.e. changing/keeping variables' values since the previous time moment. In this paper, we focus on LTL semantics of EDTL-requirements, hence we consider EDTL-formulas are constructed from propositions as follows:

Definition 2. *(EDTL-formulas)*
Let p be a proposition, φ and ψ be EDTL-formulas. Then:

- state formulas:
  - $true$, $false$, and p are an atomic EDTL-formulas;
  - $\varphi \wedge \psi$ is the conjunction of φ and ψ;
  - $\varphi \vee \psi$ is the disjunction of φ and ψ;
  - $\neg\varphi$ is the negation of φ;
- event formulas with proposition[3] p:
  - $\backslash p$ is the falling edge: the value of p changes from $false$ to $true$;
  - $/p$ is the rising edge: the value of p changes from $true$ to $false$;
  - $\_p$ is low steady-state: the value of p remains equal to $false$;
  - $\sim p$ is high steady-state: the value of p remains equal to $true$.

The detailed semantics of these formulas is given in [16]. Informally, the semantics of the state formulas is standard semantics for LTL-formulas without temporal operators, but the semantics of the event formulas uses satisfiability of the proposition in the previous system state. This semantics can be modelled by temporal operator $\mathbf{X}^{-1}$ of PLTL [8] or by introducing into the model special ghost variable $prev(p)$ which keep the previous value of proposition p. In the latter case, the semantics of the event formulas is reduced to semantics of the state EDTL-formulas as follows:

[3] In general, event formulas can take all EDTL-formulas, but for simplicity, we restrict them by propositions only. The results of this paper can easily be generalised for all EDTL-formulas.

Definition 3. *(Semantics of event EDTL-formulas)*

- $/p \equiv \neg prev(p) \wedge p$;
- $\backslash p \equiv prev(p) \wedge \neg p$;
- $\sim p \equiv prev(p) \wedge p$;
- $\_p \equiv \neg prev(p) \wedge \neg p$.

Further in the paper, we assume that all EDTL-formulas are state formulas.

Let us define the LTL-semantics for EDTL-requirements. In this paper, we consider control systems which have model as a Kripke structure. Let P be set of propositions.

Definition 4. *(Kripke structures)*
A Kripke structure is a tuple $M = (S, I, R, L)$, where

- S is a set of states, and
- $I \subseteq S$ is a finite set of initial states, and
- $R \subseteq S \times S$ is a total transition relation, and
- $L : P \to 2^S$ is a mapping function.

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

Let for control system C its Kripke structure be M_C. Let *trigger*, *invariant*, $final$, *delay*, *reaction*, and *release* be EDTL-formulas which are the values of the EDTL-attributes of a requirement *req*.

Definition 5. *(Semantics of EDTL-requirements)*
EDTL-requirement req is satisfied in a control system C iff the following LTL-formula Φ_{req} is satisfied in M_C for every initial path:
$\Phi_{req} = \mathbf{G}(trigger \to ((invariant \wedge \neg final \mathbf{W} release) \vee$
$(invariant \mathbf{U}(final \wedge (invariant \wedge delay \mathbf{U}(release \vee reaction))))))$.

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

Definition 6. *(Satisfiability of EDTL-requirements)*
Requirement r is *satisfiable* iff there exists a Kripke structure M_r that for every initial path π: $M_r, \pi \models \Phi_r$. This M_r is a *model* for r.

3 Consistency of EDTL-requirements

In this section, we give the method of checking consistency for two requirements which patterns have non-constant attributes only.

Definition 7. *(The checking inconsistency problem for EDTL-requirements)*
Requirement r_2 is *inconsistent* with satisfiable requirement r_1 iff M_{r_1} is not a model for r_2, or, equivalent, their conjunction is unsatisfiable formula in every model M_{r_1}, i.e. there exists an initial path π of M_{r_1}: $M_{r_1}, \pi \not\models \Phi_{r_1} \wedge \Phi_{r_2}$.
The checking inconsistency problem for EDTL-requirements is to check if two EDTL-requirements are inconsistent.

We use the second form of the inconsistency definition in our checking method because we compare only the requirements' attributes without addressing the models of the requirements and paths in these models. Moreover, due to this independence from the requirement model, we can deduce that $M_{r_1}, \pi \not\models \Phi_{r_1} \wedge \Phi_{r_2}$ for *every* initial path π of M_{r_1} if r_2 is inconsistent with r_1.

From Definition 5, the requirement semantics is $\Phi_{req} = \mathbf{G}(trigger \rightarrow \Psi)$. Let we be given two EDTL-requirements r_1 and r_2 with corresponding patterns $R_1 = (trig_1, inv_1,\ fin_1, del_1, rea_1, rel_1)$ and $R_2 = (trig_2, inv_2, fin_2, del_2,\ rea_2, rel_2)$. Hence, for every initial path of every Kripke structure M: $M, \pi \not\models \Phi_{r_1} \wedge \Phi_{r_2}$ iff $M, \pi \models \neg(\Phi_{r_1} \wedge \Phi_{r_2})$. Here $\neg(\Phi_{r_1} \wedge \Phi_{r_2}) = \neg(\mathbf{G}(trig_1 \rightarrow \Psi_1) \wedge \mathbf{G}(trig_2 \rightarrow \Psi_2)) = \neg\mathbf{G}((trig_1 \rightarrow \Psi_1) \wedge (trig_2 \rightarrow \Psi_2))$. Our consistency checking procedure *Compare* is based on assumption that $trig_1 \rightarrow trig_2$. This assumptions gives us possibility to reason about simultaneous satisfiability of EDTL-attributes at some point of a model path in many cases described below because the last formula implies $\neg\mathbf{G}(trig_1 \rightarrow (\Psi_1 \wedge \Psi_2))$. Definitely, with this weakening inconsistency condition, the results of our algorithm are partial, i.e. there exists inconsistent req_1 and req_2 for which procedure *Compare* gives the output "unknown". For checking such requirements, an explicit description of a control software model or an automata-based satisfiability checking method is required.

The following reasoning is based on assumptions that (1) $trig_1 \rightarrow trig_2$ and (2) Φ_{r_1} is satisfiable in some model M_{r_1}, i.e. for every initial path π of M_{r_1}, $M_{r_1}, \pi \models \Phi_{r_1}$. By Definition 7 with the weak inconsistency condition $M_{r_1}, \pi \models \neg\mathbf{G}(trig_1 \rightarrow (\Psi_1 \wedge \Psi_2))$ and assumption (2), r_2 is inconsistent with r_1 iff there exist initial path π' of M_{r_1} such that $M_{r_1}, \pi' \models \mathbf{F}\neg(trig_1 \rightarrow \Psi_2)$, i.e. there is some point s' on π' where $\neg(trig_1 \rightarrow \Psi_2)$ holds on suffix $\pi'_{s'}$ of π'. We can use this fact to describe inconsistency restrictions for r_2. But we also know that $M_{r_1}, \pi'_{s'} \models trig_1 \rightarrow \Psi_1$ due to assumption (2), hence, $M_{r_1}, \pi'_{s'} \models \Psi_1 \rightarrow \neg\Psi_2$. Therefore, we consider two cases to describe some inconsistency restrictions for r_2 which do not require knowledge about model M_{r_1}: (1) $\neg(trig_1 \rightarrow \Psi_2)$ in Subsection 3.1 and (2) $\Psi_1 \rightarrow \neg\Psi_2$ in Subsection 3.2.

Let φ_1 and φ_2 be EDTL-attributes which conjunction is false. We call φ_1 and φ_2 *inconsistent EDTL-formulas* and use the following notation: $\varphi_1 \bullet \varphi_2$ iff $\varphi_1 \wedge \varphi_2 \equiv false$, and $\varphi_1 \circ \varphi_2$ in other case. For any pair of EDTL-formulas we can check their inconsistency using standard boolean rules and Definition 3. Obviously, this checking can be reduced to NP-complete SAT problem. But due to the small size of EDTL-formulas, check their inconsistency takes reasonable time. In the description of inconsistency restrictions, for every attributes a_1 and a_2, we suppose that if not $a_1 \bullet a_2$ then $a_1 \circ a_2$.

The values of EDTL-attributes of requirements can be constant $false/true$ or mutable var. In this section, we consider the most general case when every attribute of R_1 and R_2 has non-constant value.

3.1 Inconsistency Restrictions with Attribute *trigger*

In this subsection, we describe inconsistency restriction for r_2 which provide $\neg(trig_1 \rightarrow \Psi_2) = \neg(trig_1 \wedge \Psi_2)$ on path $\pi'_{s'}$. By assumption (2), $trig_1$ holds

at s'. From Definition 5, $\Psi_2 = (inv_2 \wedge \neg fin_2 \mathbf{W} rel_2) \vee (inv_2 \mathbf{U}(fin_2 \wedge (inv_2 \wedge del_2 \mathbf{U}(rel_2 \vee rea_2))))$. The following restriction on EDTL-attributes of r_1 and r_2 implies $\neg(trig_1 \wedge \Psi_2)$:

1. $trig_1 \bullet inv_2$ and $C_2 = \neg((trig_1 \vee inv_1 \vee trig_2) \rightarrow (rel_2 \vee (fin_2 \wedge rea_2)))$;
2. other restrictions are unknown.

Because Ψ_2 includes temporal operators and $trig_1$ do not include them, we can reason only about first point s' of the path on which $trig_1$ holds and Ψ_2 does not hold. The negation of C_2 cancels necessity of satisfiability of inv_2. Hence, in this case, its inconsistency with $trig_1$ does not matter for inconsistency of r_2.

3.2 Inconsistency Restrictions with Other Attributes

In this subsection, we describe inconsistency restrictions for r_2 which provide $\Psi_1 \rightarrow \neg\Psi_2$ at $\pi'_{s'}$. From Definition 5, $\Psi_1 = A_1 \vee B_1$ and $\Psi_2 = A_2 \vee B_2$. Hence, $\Psi_1 \rightarrow \neg\Psi_2 = \neg(A_1 \vee B_1) \vee \neg(A_2 \vee B_2)$. Due to satisfiability of Φ_{r_1}, $A_1 \vee B_1$ is satisfiable. Hence, we consider the cases $A_1 \rightarrow \neg(A_2 \vee B_2)$ and $B_1 \rightarrow \neg(A_2 \vee B_2)$. These cases are also divided to subcases: (AA) $A_1 \rightarrow \neg A_2$, (AB) $A_1 \rightarrow \neg B_2$, (BA) $B_1 \rightarrow \neg A_2$, and (BB) $B_1 \rightarrow \neg B_2$. We formulate inconsistency restrictions for all these cases. Again, from Definition 5:

- $A_1 = (inv_1 \wedge \neg fin_1 \mathbf{W} rel_1)$,
- $B_1 = (inv_1 \mathbf{U}(fin_1 \wedge (inv_1 \wedge del_1 \mathbf{U}(rel_1 \vee rea_1))))$,
- $A_2 = (inv_2 \wedge \neg fin_2 \mathbf{W} rel_2)$,
- $B_2 = (inv_2 \mathbf{U}(fin_2 \wedge (inv_2 \wedge del_2 \mathbf{U}(rel_2 \vee rea_2))))$.

(AA) $A_1 \rightarrow \neg A_2$**.** A_1 is satisfiable on path $\pi'_{s'}$. The following restrictions on EDTL-attributes of r_1 and r_2 imply $\Psi_1 \rightarrow \neg\Psi_2$:

1. $inv_1 \bullet inv_2$, $C_1 = \neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow rel_1)$, and $C_2 = \neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow rel_2)$;
2. $\neg fin_1 \bullet inv_2$, and $C_1 \wedge C_2$;
3. $inv_1 \bullet \neg fin_2$, and $C_1 \wedge C_2$;
4. $\neg fin_1 \bullet \neg fin_2$, and $C_1 \wedge C_2$;
5. $rel_1 \bullet inv_2$, and $\neg C_1 \wedge C_2$;
6. $rel_1 \bullet \neg fin_2$, and $\neg C_1 \wedge C_2$;
7. $rel_1 \bullet rel_2$, and $\neg C_1 \wedge \neg C_2$;
8. other restrictions are unknown.

In all cases, $trig_1$ and $trig_2$ hold at s', inv_1 and $\neg fin_1$ also hold at s', if only rel_1 does not happen at this point. Let all restrictions of every case hold at s' separately. Formulas C_1 and C_2 provides that, respectively, releases rel_1 or rel_2 do not happen at s', because their appearing makes senseless inconsistency of the corresponding attributes. From now, formulas C_1 and C_2 specify cancelling conditions respective to considering cases of restrictions.
(1-2) Obviously, in these cases, $A_1 \rightarrow \neg inv_2$ holds, which implies unsatisfiability of A_2 at s'.

(3-4) In these cases, the reasoning is the same.
(5-7) Restrictions of these cases specify appearing rel_1 at s' and its inconsistency with attributes of r_2. Definitely, in these cases, $A_1 \rightarrow \neg(inv_2 \vee \neg fin_2 \vee rel_2)$ holds which implies unsatisfiability of A_2 at s'.
(8) Our inconsistency checking algorithm checks EDTL-attributes and their Boolean combinations which are state formulas. Hence, this algorithm can detect inconsistency of EDTL-requirements in model states only. In case (AA), there is only one state s' at which we know about satisfiability of r_1-attributes and can define inconsistency restrictions for r_2-attributes. Hence, all possibilities for defining static inconsistency restrictions are considered in cases 1–7, and for defining other inconsistency restrictions, the description of model M_{r_1} is required.
(AB) $A_1 \rightarrow \neg B_2$**.** A_1 is satisfiable on path $\pi'_{s'}$. The following restrictions on EDTL-attributes of r_1 and r_2 imply $\Psi_1 \rightarrow \neg\Psi_2$:

1. $inv_1 \bullet inv_2$, $C_1 = \neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow rel_1)$, and $C_2 = \neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow (rel_2 \vee (fin_2 \wedge rea_2)))$;
2. $\neg fin_1 \bullet inv_2$, and $C_1 \wedge C_2$;
3. $inv_1 \rightarrow fin_2$, $\neg fin_1 \bullet del_2$, $(inv_1 \vee \neg fin_1) \bullet rel_2$, $(inv_1 \vee \neg fin_1) \bullet rea_2$, and C_1;
4. $\neg fin_1 \rightarrow fin_2$, $inv_1 \bullet del_2$, $(inv_1 \vee \neg fin_1) \bullet rel_2$, $(inv_1 \vee \neg fin_1) \bullet rea_2$, and C_1;
5. $rel_1 \bullet inv_2$, and $\neg C_1 \wedge C_2$;
6. other restrictions are unknown.

In all cases, $trig_1$ and $trig_2$ hold at s', inv_1 and $\neg fin_1$ also hold at s', if only rel_1 does not happen at this point. Let all restrictions of every case hold at s' separately.
(1-2) In these cases, $A_1 \rightarrow \neg inv_2$, hence $A_1 \rightarrow \neg B_2$.
(3-4) Here, inv_1 (or $\neg fin_1$) causes fin_2, and inconsistency $\neg fin_1$ (or inv_1) with del_2 requires immediate release rel_2 or reaction rea_2, but they are also inconsistent with inv_1 or $\neg fin_1$. Hence, $A_1 \rightarrow \neg(fin_2 \wedge (inv_2 \wedge del_2 \mathbf{U}(rel_2 \vee rea_2)))$, which implies unsatisfiability of B_2 at s'. Note, that an event which causes some final event cannot be inconsistent with the corresponding delay because this delay starts from this final event which implies that *delay* must hold at the state where $final$ happens after the corresponding $trigger$.
(5) Restrictions of these cases specify appearing rel_1 at s' and its inconsistency with attributes of r_2 similarly to cases 1–4.
(6) As for the case (AA), there is only one state s' at which we know about satisfiability of r_1-attributes. Hence, for defining inconsistency restrictions different from 1–5, the description of model M_{r_1} is required.
(BA) $B_1 \rightarrow \neg A_2$**.** B_1 is satisfiable on path $\pi'_{s'}$. The following restrictions on EDTL-attributes of r_1 and r_2 imply $\Psi_1 \rightarrow \neg\Psi_2$:

1. $inv_1 \bullet inv_2$, and $C_1 = \neg(C_1^1 \vee C_1^2)$, and $C_2 = \neg((trig_1 \vee inv_1 \vee trig_2) \rightarrow rel_2)$, where $C_1^1 = \neg((trig_1 \vee inv_1 \vee trig_2) \rightarrow rel_1)$ and $C_1^2 = \neg((trig_1 \vee inv_1 \vee trig_2) \rightarrow (fin_1 \wedge rea_1))$;

2. $inv_1 \bullet \neg fin_2$, and $C_1 \wedge C_2$;
3. $rel_1 \bullet inv_2$, and $\neg C_1^1 \wedge C_2$;
4. $rel_1 \bullet \neg fin_2$, and $\neg C_1^1 \wedge C_2$;
5. $(fin_1 \wedge rea_1) \bullet inv_2$, and $\neg C_1^2 \wedge C_2$;
6. $(fin_1 \wedge rea_1) \bullet \neg fin_2$, and $\neg C_1^2 \wedge C_2$;
7. other restrictions are unknown.

In all cases, $trig_1$ and $trig_2$ hold at s', and inv_1 also hold at s', if only rel_1 or fin_1 with rea_1 do not happen at this point, and there exists states s'_1 and s'_2 on path π' after s' at which $fin_1 \wedge inv_1$ and $rel_1 \vee rea_1$ hold respectively. Let all restrictions of every case hold at s' separately.
(1-2) In both restriction cases, $B_1 \to \neg inv_2$, hence $B_1 \to \neg A_2$.
(3-6) These cases describes inconsistency restrictions when release rel_1 or final fin_1 with immediate reaction rea_1 cancel invariant inv_1 and contradict r_2-attributes. All these restrictions obviously imply $B_1 \to \neg A_2$.
(7) Surprisingly, although we know that there exists states s'_1 and s'_2, we cannot decide about inconsistency of rel_1, fin_1, and rea_1 respective to neither inv_2, nor $\neg fin_2$ because appearing rel_2 may happen at any π'-path state between s' and s'_1/s'_2. Formula that specify this appearance is $\neg(\mathbf{F}(rel_1 \vee (fin_1 \wedge rea_1)) \wedge (\neg rel_2 \mathbf{U}(rel_1 \vee (fin_1 \wedge rea_1))))$. The negation of this formula could be the sufficient condition for inconsistency restrictions combining rel_1, fin_1, rea_1 with inv_2 and $\neg fin_2$, but it is not static EDTL-formula and its checking requires the description of model M_{r_1}.
(BB) $B_1 \to \neg B_2$**.** B_1 is satisfiable on path $\pi'_{s'}$. The following restrictions on EDTL-attributes of r_1 and r_2 imply $\Psi_1 \to \neg\Psi_2$:

1. $inv_1 \bullet inv_2$, and $C_1 = \neg(C_1^1 \vee C_1^2)$, and $C_2 = \neg((trig_1 \vee inv_1 \vee trig_2) \to (rel_2 \vee (fin_2 \wedge rea_2))$, where $C_1^1 = \neg((trig_1 \vee inv_1 \vee trig_2) \to rel_1)$ and $C_1^2 = \neg((trig_1 \vee inv_1 \vee trig_2) \to (fin_1 \wedge rea_1))$;
2. $rel_1 \bullet inv_2$, and $\neg C_1^1 \wedge C_2$;
3. $(fin_1 \wedge rea_1) \bullet inv_2$, and $\neg C_1^2 \wedge C_2$;
4. $rel_1 \to fin_2$, $rel_1 \bullet inv_2$, and $\neg((rel_1 \vee fin_2) \to (rel_2 \vee rea_2))$;
5. $rea_1 \to fin_2$, $rea_1 \bullet inv_2$, and $\neg((rea_1 \vee fin_2) \to (rel_2 \vee rea_2))$;
6. $fin_1 \to fin_2$, $fin_1 \bullet inv_2$, and $\neg((fin_1 \vee fin_2) \to (rel_2 \vee rea_2))$;
7. $fin_1 \to fin_2$, $inv_1 \bullet del_2$, $(inv_1 \vee fin_1) \bullet rel_2$, and $(inv_1 \vee fin_1) \bullet rea_2$;
8. other restrictions are unknown.

In all cases, $trig_1$ and $trig_2$ hold at s', and inv_1 also hold at s', if only rel_1 or fin_1 with rea_1 do not happen at this point, and there exists states s'_1 and s'_2 on path π' after s' at which $fin_1 \wedge inv_1$ and $rel_1 \vee rea_1$ hold respectively. Let (1) all restrictions of cases 1–3 hold at s', (2) all restrictions of cases 4-5 hold at s'_2, and (3) all restrictions of cases 6-7 hold at s'_1.
(1-3) This cases are similar to the case (BA).
(4-6) This cases exploit the fact that inv_2 must hold at the state where fin_2 holds if only there is no release rel_2 or reaction rea_2. In cases 4 and 5, this state is s'_2, and in case 6, it is s'_1.

(7) In this case, similarly to case (AB-3), fin_1 causes fin_2 at state s'_1, and inconsistency of inv_1 with del_2 requires immediate release rel_2 or reaction rea_2, but they are also inconsistent with $inv_1 \vee fin_1$.
(8) We consider all possible restrictive combinations for attributes of requirements r_1 and r_2 which allow static inconsistency checking. Other restrictions require the description of model M_{r_1}.

The restrictions on EDTL-attributes of r_1 and r_2 formulated above implies unsatisfiability of $\Phi_{r_1} \wedge \Phi_{r_2}$ in model M_{r_1}. Hence, if the requirements satisfy these restrictions, our algorithm yields definite result of inconsistency checking. Summarise these restrictions in the next section.

4 Procedure *Compare*

Our goal is to describe a function which solves if two EDTL-requirements (in)-consistent using the attribute values of their patterns only. Let $Compare(R_1, R_2)$ be a function which input is patterns and output is answer from the set {*consistent*, *inconsistent*, *unknown*}.

To compare the requirements, we use the following method. First, using the results of the previous section, we define answers and their conditions of function *Compare* for the most general case when every attribute of R_1 and R_2 has non-constant value. For attribute constant values, we substitute them into the general form of answer to get the answers and their conditions for these particular cases.

In the following definition of the outputs of $Compare(R_1, R_2)$, we summarise the restrictions on the attributes of requirement r_2 which cause its inconsistency with requirement r_1. In this definition, conjunction of all cases of each output marked by letters gives its necessary condition.

1. $Compare(R_1, R_2) = inconsistent$, if
   (a) $inv_1 \bullet inv_2$;
   (b) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow (rel_1 \vee (fin_1 \wedge rea_1))$;
   (c) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow (rel_2 \vee (fin_2 \wedge rea_2)))$.
2. $Compare(R_1, R_2) = inconsistent$, if
   (a) $\neg fin_1 \bullet inv_2$;
   (b) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow rel_1)$;
   (c) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow (rel_2 \vee (fin_2 \wedge rea_2))$.
3. $Compare(R_1, R_2) = inconsistent$, if
   (a) $inv_1 \bullet \neg fin_2$;
   (b) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow (rel_1 \vee (fin_1 \wedge rea_1))$;
   (c) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow rel_2)$.
4. $Compare(R_1, R_2) = inconsistent$, if
   (a) $\neg fin_1 \bullet \neg fin_2$;
   (b) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow rel_1)$;
   (c) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow rel_2)$.
5. $Compare(R_1, R_2) = inconsistent$, if
   (a) $rel_1 \bullet inv_2$;
   (b) $(trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \rightarrow rel_1$;

   (c) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \to (rel_2 \vee (fin_2 \wedge rea_2))$.
6. $Compare(R_1, R_2) = inconsistent$, if
   (a) $rel_1 \bullet \neg fin_2$;
   (b) $(trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \to rel_1$;
   (c) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \to rel_2)$.
7. $Compare(R_1, R_2) = inconsistent$, if
   (a) $rel_1 \bullet rel_2$;
   (b) $(trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \to rel_1$;
   (c) $(trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \to rel_2$.
8. $Compare(R_1, R_2) = inconsistent$, if
   (a) $inv_1 \to fin_2$;
   (b) $\neg fin_1 \bullet del_2$;
   (c) $(inv_1 \vee \neg fin_1) \bullet rel_2$;
   (d) $(inv_1 \vee \neg fin_1) \bullet rea_2$;
   (e) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \to rel_1)$.
9. $Compare(R_1, R_2) = inconsistent$, if
   (a) $\neg fin_1 \to fin_2$;
   (b) $inv_1 \bullet del_2$;
   (c) $(inv_1 \vee \neg fin_1) \bullet rel_2$;
   (d) $(inv_1 \vee \neg fin_1) \bullet rea_2$;
   (e) $\neg((trig_1 \vee inv_1 \vee \neg fin_1 \vee trig_2) \to rel_1)$.
10. $Compare(R_1, R_2) = inconsistent$, if
   (a) $(fin_1 \wedge rea_1) \bullet inv_2$;
   (b) $(trig_1 \vee inv_1 \vee trig_2) \to (fin_1 \wedge rea_1)$;
   (c) $\neg((trig_1 \vee inv_1 \vee trig_2) \to (rel_2 \vee (fin_2 \wedge rea_2))$.
11. $Compare(R_1, R_2) = inconsistent$, if
   (a) $(fin_1 \wedge rea_1) \bullet \neg fin_2$;
   (b) $(trig_1 \vee inv_1 \vee trig_2) \to (fin_1 \wedge rea_1)$;
   (c) $\neg((trig_1 \vee inv_1 \vee trig_2) \to rel_2)$.
12. $Compare(R_1, R_2) = inconsistent$, if
   (a) $rel_1 \to fin_2$;
   (b) $rel_1 \bullet inv_2$;
   (c) $\neg((rel_1 \vee fin_2) \to (rel_2 \vee rea_2))$.
13. $Compare(R_1, R_2) = inconsistent$, if
   (a) $rea_1 \to fin_2$;
   (b) $rea_1 \bullet inv_2$;
   (c) $\neg((rea_1 \vee fin_2) \to (rel_2 \vee rea_2))$.
14. $Compare(R_1, R_2) = inconsistent$, if
   (a) $fin_1 \to fin_2$;
   (b) $fin_1 \bullet inv_2$;
   (c) $\neg((fin_1 \vee fin_2) \to (rel_2 \vee rea_2))$.
15. $Compare(R_1, R_2) = inconsistent$, if
   (a) $fin_1 \to fin_2$;
   (b) $inv_1 \bullet del_2$;
   (c) $(inv_1 \vee fin_1) \bullet rel_2$;
   (d) $(inv_1 \vee fin_1) \bullet rea_2$.

16. $Compare(R_1, R_2) = consistent$, if
    (a) $inv_1 \rightarrow inv_2$;
    (b) $fin_1 \rightarrow fin_2$;
    (c) $del_1 \rightarrow del_2$;
    (d) $rea_1 \rightarrow rea_2$;
    (e) $rel_1 \rightarrow rel_2$;
17. $Compare(R_1, R_2) = unknown$, other cases.

All outputs above are based on results of Section 3 except case 16. In this case, weakening every attribute of R_1 in R_2 implies satisfiability of r_2 in every model where r_1 is satisfiable.

Outputs of *Compare* for the other combinations of attribute values of R_1 and mutable attribute values of R_2 can be found in [18].

5 Algorithm for checking consistency of EDTL-Patterns

Let a given set of requirements be presented as EDTL-patterns. The checking consistency algorithm *Consistency_Checker* compares the requirements in pairs using the function *Compare*. For each requirement the algorithm generates a list of inconsistent, consistent and undefined requirements. Note that the inconsistency relation is not transitive. Using these lists, we can compile sets of consistent requirements, as well as lists of requirements, whose consistency should be further verified by stronger methods. The complexity of this algorithm is quadratic with respect to the size of the set of requirements.

```
type Req :
     struct {
       pattern : array [6] of EDTL_formula; // 0-trig, ..., 5-rel
       inconsistent : list of Req;
       consistent : list of Req;
       unknown : list of Req;
}

Consistency_Checker (reqs : array [n] of req){
for i = 1 .. n-2
    for j = i+1 .. n
        res = Decide( reqs[i], reqs[j]);
        case (res) {
              inconsistent : reqs[i].inconsistent.add(reqs[j]);
                             reqs[j].inconsistent.add(reqs[i]);
              consistent : reqs[i].consistent.add(reqs[j]);
                           reqs[j].consistent.add(reqs[i]);
              unknown : reqs[i].unknown.add(reqs[j]);
                        reqs[j].unknown.add(reqs[i]);
        }
}
```

First, the function *Decide* checks comparability of requirements ($trig_1 \to trig_2$ or $trig_2 \to trig_1$), which takes exponential time with respect to the size of the triggers. Then it tries if the semantics of incoming requirements is *true* or $false$ with partial function $Compute\_semantics$ which substitute the attribute values to LTL semantic formula, and returns 'true'/'false' iff the result of the substitution is identically true or false, and 'unknown' in other cases. The time complexity of this function is linear with respect to the size of the requirements. Its definition is trivial and out of the scope of this paper. If there are the cases *true* or $false$, it returns values to $Consistency\_Checker$ immediately. If not, it calls function *Compare* which compute if the requirements inconsistent.

```
Decide (pat1, pat2){
 if !imply(pat1[0], pat2[0]) && !imply(pat2[0], pat1[0]) &&
                                !pat1[0] && !pat2[0]
                                            then return unknown;
 res = Compute_semantics( pat1 );
 if res = 'true' then return unknown;
 if res = 'false' then return inconsistent;
 res = Compute_semantics( pat2 );
 if res = 'true' then return unknown;
 if res = 'false' then return inconsistent;
 if imply(pat1[0], pat2[0]) then return Compare( pat1, pat2 );
 if imply(pat2[0], pat1[0]) then return Compare( pat2, pat1 );
}
```

The function *Compare* is based on its definition given the previous section. For example, the following part of the code of *Compare* models case (13) of this definition. The function $Compute(frm)$ returns *true* iff frm is identically true formula. The definition of this function is based on standard Boolean rules and Definition 3. The complexity of this function is exponential with respect to the size of the attributes of requirements because it solves SAT problem.

```
Compare (pat1, pat2){
           ...
           // case 13
           if Compute((pat1[4] -> pat2[2]) &&
                     !(pat1[4] && pat2[1]) &&
                     !((pat2[2]||pat1[4])->(pat2[4]||pat2[5])))
                                        then return inconsistent;
           ...
           return unknown;
}
```

Using the mentioned above time complexities of the algorithm and its functions, we state the following

Theorem 1. *There exists the algorithm partially solving the checking inconsistency problem for EDTL-requirements which takes quadratic time with respect to*

the size of the set of requirements and exponential time with respect to the size of the requirements.

Standard automata-based satisfiability checking algorithms for LTL-formula φ take exponential time $T_s(\varphi)$ with respect to the size of the checking formula. Roughly, if the size of every EDTL-attribute of each EDTL-requirement is a then $T_s(\Phi_{r_1} \wedge \Phi_{r_2}) \geq 2^{20*a}$. The time complexity $T_d(r_1, r_2)$ of the most expensive function *Decide* is the sum of complexities of the first line checking and the *Compare* function: $T_d(r_1, r_2) \leq 2^{2*a} + 2^{2*a+4} + 2^{7*a+5}$. Definitely, T_s is always greater than T_d for a pair of EDTL-requirements independently their size[4]. Hence, our simple checking algorithm can be used before more powerful and complex automata-based checking inconsistency.

6 Conclusion

In this paper, we propose the method and algorithm for checking the consistency of requirements presented as EDTL-patterns. This method uses LTL semantics of requirements. It analyses key cases for combinations of pattern attribute values. We present an exhaustive description of these cases to determine that two requirements are not consistent if the attribute values of their patterns are not Boolean constants. For other attribute values, the method suggests substituting them into the described inconsistency conditions to obtain the conditions corresponding to these values. We describe the pseudocodes of algorithms that implement the proposed method. The results of the algorithm can be used to compile sets of consistent requirements, as well as lists of requirements for consistency checking by stronger methods. The complexity of the main algorithm is quadratic with respect to the size of the set of requirements. The complexity of the comparison function used in the algorithm is exponential with respect to the size of the attributes. Nevertheless, we show that the overall complexity of our algorithm is much better than the complexity of automata-based satisfiability checking algorithms for LTL-formulas. Hence, incompleteness of results of our algorithm is balanced by its relatively low complexity.

This study is a part of theoretical and practical research on the development and verification of process-oriented control software [1,9,16]. The results of the paper will be used in a general approach to checking the consistency of EDTL-requirements. For this approach, we currently develop methods that explicitly use the temporal semantics of requirements. One of them is the construction of equivalent finite automata for semantics of EDTL-requirements and checking their consistency.

References

1. Anureev, I.S. Operational Semantics of Annotated Reflex Programs. Aut. Control Comp. Sci. 54, 719–727 (2020). https://doi.org/10.3103/S0146411620070032

[4] For example, if $a = 1$, then $T_s \geq 1048576$ and $T_d = 1332$.

2. 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 Transactions on Software Engineering*, vol.41, no.7, pp. 620–638, 2015.
3. V. Bashev, I. Anureev and V. Zyubin, "The Post Language: Process-Oriented Extension for IEC 61131-3 Structured Text," 2020 International Russian Automation Conference (RusAutoCon), Sochi, Russia, 2020, pp. 994-999, doi: 10.1109/RusAutoCon49822.2020.9208049.
4. Clarke, E.M., Henzinger, Th.A., Veith, H., Bloem, R. (Eds.): Handbook of Model Checking. Chapter 18. Springer International Publishing (2018).
5. Dwyer,M.B., Avrunin,G.S., Corbett, J.C. “Patterns in property specifications for finite-state verification", *in Proc. of the 21st Int. Conf. on Software Engineering*, IEEE Computer Society Press, 1999, pp. 411–420.
6. N. Garanina, I. Anureev, E. Sidorova, D. Koznov, V. Zyubin, and S. Gorlatch. An Ontology-based Approach to Support Formal Verification of Concurrent Systems // Formal Methods. FM 2019 International Workshops. LNCS Volume 12232, pp. 114-130 https://doi.org/10.1007/978-3-030-54994-7_9
7. N. Garanina and O. Borovikova, "Ontological Approach to Checking Event Consistency for a Set of Temporal Requirements," 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON), Novosibirsk, Russia, 2019, pp. 0922-0927, doi: 10.1109/SIBIRCON48586.2019.8958119.
8. Heljanko, K., Junttila, T.A., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Etessami, K., Rajamani, S.K. (eds.) Intl. Conf. on Computer-Aided Verification (CAV). LNCS, vol. 3576, pp. 98–111. Springer, Heidelberg (2005)
9. Liakh T.V., Rozov A.S., Zyubin V.E.: Reflex Language: a Practical Notation for Cyber-Physical Systems. System Informatics **12**, 85–104 (2018)
10. Mondragon, O., Gates, A.Q., Roach, S. “Prospec: Support for Elicitation and Formal Specification of Software Properties", *in Proc. of Runtime Verification Workshop of Electronic Notes in Theoretical Computer Science*, 2014, vol. 89, pp. 67–88.
11. Salamah, S., Gates, A.Q., Kreinovich, V. “Validated patterns for specification of complex LTL formulas", *Journal of Systems and Software*, 85(8), pp.1915–1929, 2012.
12. Simko, V., Hauzar, D., Bures, T., Hnetynka, P., Plasil, F. “Verifying Temporal Properties of Use-Cases in Natural Language", in *Proc. of Formal Aspects of Component Software. FACS 2011.*, LNCS, Springer, Berlin, 2012, vol. 7253, pp.350–367.
13. Smith, M., Holzmann, G., Etessami, K. “Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs", *in Proc. of 5 IEEE International Symposium on Requirements Engineering*, Aug. 27-31, 2001, pp. 14–22.
14. Wong, P.Y.H., Gibbons, J. “Property Specifications for Workflow Modelling", *in Proc. of Integrated Formal Methods (IFM 2009)*, LNCS, Springer-Verlag: Berlin, vol. 5423, pp. 166–180, 2009.
15. Yu, T.P., Manh, J. Han et al. “Pattern based property specification and verification for service composition", *in Proc. of 7th International Conference on Web Information Systems Engineering (WISE)*, LNCS, Springer-Verlag: Berlin, vol. 4255, pp. 156–168, 2006.
16. Vladimir Zyubin, Igor Anureev, Natalia Garanina, Sergey Staroletov, Andrei Rozov, and Tatiana Liakh. Event Driven Temporal Logic for Control Software Requirements // Proc. Of 9th IPM International Conference on Fundamentals of Software Engineering (FSEN 2021), LNCS, Springer. Accepted to publication. http://fsen.ir/2021/Programme.aspx

17. Argosim Homepage, www.argosim.com. Last accessed 20 Apr 2021
18. Garanina, N.: EDTL: Checking consistency tables.
https://github.com/GaraninaN/CheckEDTL (2021) Accessed 20 Apr 2021