Specifying Event-Based Systems with a Counting Fluent Temporal Logic

Germán Regis† Renzo Degiovanni‡ Nicolas D’Ippolito† Nazareno Aguirre‡
†Departamento de Computation, Universidad Nacional de Río Cuarto, Argentina
‡Departamento de Computación, Universidad Nacional de Río Cuarto, Argentina
(rdegiovanni, naguirre)@dc.exa.unrc.edu.ar
†Department of Computing, Imperial College London, UK
srdipi@doc.ic.ac.uk
‡Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

ABSTRACT
Event-based formalisms have been shown suitable for different specification settings in software engineering. Fluents have been introduced in order to capture state propositions in event-based systems, in terms of activating and deactivating events. Since temporal logics use atomic propositions over system states as the basis for formulas, fluents enable using these logics for property specification in event-based systems.

In this paper, we introduce counting fluent temporal logic, an extension of fluent linear temporal logic, which complements the notion of fluent by the related concept of counting fluent. As opposed to the binary nature of fluents (which are logical propositions), counting fluents are numerical values, that enumerate event occurrences, and allow one to specify naturally some properties of reactive systems, such as bounded liveness and other properties in which the number of occurrences of certain events is relevant. Although the introduced logic is an undecidable, strictly more expressive, extension of fluent linear temporal logic, we develop a sound (but incomplete) model checking approach for the logic, that reduces to LTSA model checking, enabling one to automatically verify counting fluent temporal logic properties of finite event-based systems, under user defined bounds for counting fluents. We illustrate the benefits of the introduced formalism for specifying properties of event-based systems, by comparing it with traditional temporal logic specifications of relevant models taken from the literature.

1. INTRODUCTION
The increasingly rich set of tools and techniques for software analysis offers unprecedented opportunities for helping software developers in finding program bugs, and discovering flaws in software models. An essential part of these tools and techniques is the formal specification of software properties. Various formalisms and approaches have been proposed to specify properties of different kinds of systems.

In particular, temporal logic has gained significant acceptance as a vehicle for specifying properties of various kinds of software systems, most notably parallel and concurrent systems. Moreover, temporal logic has also been employed in other domains, to formally specify software requirements [19, 9], to express properties of hardware systems, and other applications.

Temporal logics are more directly applicable to system property specification when using a state based specification approach, i.e., when one is able to refer to state properties. Given the importance of event-based formalisms, such as CSP [15], CCS [30] and FSP [26], which are convenient in various specification settings in software engineering and have influenced a number of modelling languages, including concurrent system specification and architecture description languages [28, 1], some mechanisms have been proposed to capture state properties in event-based systems. A particular case, which we use as a basis in this paper, is that of fluents, introduced in [11] in order to ease the use of temporal logic for specifying properties of event-based systems. Event-based formalisms center specification around the notion of event, which is used as a means to represent components behaviour and interaction, e.g., by expressing sending or receiving messages, invoking component services, etc; fluents allow one to capture state propositions in these systems, in terms of activating and deactivating events.

Despite the success of temporal logic as a mechanism for specifying system properties, and in particular properties of event-based systems, correctly capturing software properties is still an obstacle for many developers. Even though specification patterns [12] help in this respect, by proposing patterns for properties commonly arising in practice, in many cases the inherent expressiveness of the logic makes it difficult, or even impossible, to express certain properties.

In this paper, we deal with this issue by proposing counting fluent temporal logic, an extension of fluent linear temporal logic which allows one to specify, more naturally, properties of reactive systems in which the number of occurrences of certain events is relevant. Counting fluent temporal logic complements the previously described notion of fluent by the related concept of counting fluent, which, as opposed to the binary nature of a fluent (which is a proposition), represents a numerical value that enumerates event occurrences. As we will show later on, counting fluents enable one to capture more easily some properties that often arise in reactive system specification. For instance, counting fluents allow us to easily capture system properties referring to the
difference between send and receive events, or the number of clock ticks since the last sent message, in communicating systems. They also allow us to naturally capture bounded liveness and related properties of discrete-time event systems. Such properties typically require complex nesting of temporal operators in linear temporal logic, while we conveniently capture them using counting fluents. We illustrate the benefits of the introduced formalism for specifying properties of event-based systems, by comparing it with traditional temporal logic specifications of relevant models taken from the literature. We also show that our logic favours a better separation of concerns, allowing us to express some properties that, in order to be captured in linear temporal logic, demand extensions to the system model, mixing the actual model with property related elements.

As we mentioned, an important motivation for property specification is the possibility of performing automated analysis. In fact, it is generally accepted that a convenient language for specifying system properties is not enough; such a language must be accompanied by powerful analysis mechanisms. So, appropriate automated tool support for our logic is a concern. Although the introduced logic is an undecidable, strictly more expressive, extension of fluent linear temporal logic, we develop a sound (but incomplete) model checking approach for the logic, that reduces to fluent temporal logic model checking, enabling one to automatically verify counting fluent temporal logic properties of finite event-based systems, under user defined bounds for counting fluents. Thanks to our proposed model checking approach, the above described convenience of our introduced formalism for property specification is achieved without having to fully sacrifice automated analysability.

The remainder of this article is organised as follows. Section 2 introduces preliminary concepts necessary in the paper. Section 3 presents some motivating examples that evidence the difficulties in expressing properties such as those mentioned before, and the suitability of counting fluents to ease these properties' specification. Section 4 describes Counting Fluent Temporal Logic in detail. We present the model checking approach in Section 5. Section 6 evaluates our proposed formalism, in particular assessing some relevant characteristics in comparison with linear time temporal logic, on a number of models taken from the literature. Finally, we discuss related work and present our conclusions in Section 7.

2. BACKGROUND

Labelled Transition Systems \( (LTS) \) are typically used to model the behaviour of interacting components, characterised by states and transitions between them [26]. Formally, an LTS \( P \) is a quadruple \( (Q, A, \delta, q_0) \), where \( Q \) is a finite set of states, \( A \) is the alphabet of \( P \) (a subset of the universe \( \text{Act} \) of events), \( \delta \subseteq Q \times A \cup \{\tau\} \times Q \) is a labelled transition relation, and \( q_0 \in Q \) is the initial state. The semantics of an LTS \( P \) can be defined in terms of its executions, i.e., the set of sequences of events that \( P \) can perform, starting in the initial state and following \( P \)'s transition relation.

Finite State Processes \( (FSP) \) [20] is a process algebra, whose expressions can be automatically mapped into finite LTS, and vice versa. In FSP specifications, \("\rightarrow"\) denotes event prefix, \("\mid\) denotes choice, and conditional choices can be expressed by means of \("\text{when}\) clauses. Processes may be indexed and parameterised, and can be composed in a sequential \(";\) or parallel way \("\parallel\) ".
The safety property associated with this model requires expressing that it should never be the case that red and blue cars are on the bridge at the same time. To specify this property, as put forward in [26], we need to express whether there is at least one car of each colour on the bridge. Following the solution presented in [26, Subsection 14.2.1], we take advantage of the cars identifiers (ID) per car, namely RED[ID] and BLUE[ID], to indicate whether the corresponding car is on the bridge or not. That is, for instance for red cars, we have \( \text{RED}[i].\text{ID} = \langle \text{red[i].enter} \), \text{red[i].exit} \rangle \). Then, the required safety property is specified as follows:

\[
\text{ONEWAY} = \neg (\text{RED}[1] \lor \text{RED}[2] \lor \ldots \lor \text{RED}[N]) \land \\
(\text{BLUE}[1] \lor \text{BLUE}[2] \lor \ldots \lor \text{BLUE}[N])
\]

Notice how, in this case, we are capturing the fact that there is more than one car of a given colour on the bridge through a (parameterised) disjunction, whose size depends on the number of cars allowed in each direction (often, a parameter of a bounded model abstraction of a real world situation). We will come back to this property below.

To continue our motivating example, let us suppose that we have to impose an additional constraint on the bridge model. Besides the fact that, due to the bridge’s width, cars circulating in different directions must be forbidden, assume that the bridge has a maximum weight capacity. Exceeding this capacity is dangerous, so the maximum number of cars on the bridge must also be controlled. Notice that, although this restriction was not part of the original model, such a constraint is common in these kinds of models (see, for instance, the Ornamental Garden, Bounded Buffers, Producers-Consumers, and Readers and Writers, from [26]). The controller for the bridge must now forbid new cars entering the bridge when the maximum capacity is met, which can be achieved as follows:

```
const C = 3; // maximum capacity of the bridge
BRIDGE = BRIDGE[0][0]; // initially empty
BRIDGE[nr][nb][T] = \n// nr is the red count, nb the blue count
when ((nb == 0) && (nb <= C)) \n\text{red[ID].enter} \rightarrow \text{BRIDGE}[nr+1][nb]
\| \text{red[ID].exit} \rightarrow \text{BRIDGE}[nr-1][nb]
\| \text{blue[ID].enter} \rightarrow \text{BRIDGE}[nr][nb+1]
\| \text{blue[ID].exit} \rightarrow \text{BRIDGE}[nr][nb-1].
```

Now we would like to express the fact that this controller ensures the bridge’s safety, i.e., that the number of cars on the bridge never exceeds the bridge’s capacity. We may take advantage of the previously introduced fluents that capture the fact that a particular car is on the bridge to attempt to capture this property. But, as the reader may realise, this property is more difficult to specify, since the number of possible scenarios to consider, taking into account that all interleavings of entering and leaving events have to be considered, is in principle infinite. Nevertheless, assuming that the previously specified \text{ONEWAY} property holds, we can specify the bridge’s weight safety as the following property \text{CAPACITY\_SAFE}:

\[
\text{CAPACITY\_SAFE} = \neg (\text{RED}[1] \land \text{RED}[2] \land \text{RED}[3]) \lor \\
(\text{RED}[1] \land \text{RED}[2] \land \text{RED}[4]) \lor \\
(\text{RED}[1] \land \text{RED}[3] \land \text{RED}[4]) \lor \\
(\text{RED}[2] \land \text{RED}[3] \land \text{RED}[4]) \lor \\
(\text{BLUE}[1] \land \text{BLUE}[2] \land \text{BLUE}[3]) \lor \\
(\text{BLUE}[1] \land \text{BLUE}[2] \land \text{BLUE}[4]) \lor \\
(\text{BLUE}[1] \land \text{BLUE}[3] \land \text{BLUE}[4]) \lor \\
(\text{BLUE}[2] \land \text{BLUE}[3] \land \text{BLUE}[4])
\]

As the reader may notice, this formula grows quickly as the number of cars and the bridge capacity are increased. More precisely, the number of disjunctions in this formula is in this case \( \binom{4}{3} + \binom{4}{3} = 8 \), the sum of the combinatorial numbers between the size of each convoy and the bridge’s capacity. Notice that, even for small models, this kind of property, clearly related to the need of “counting” (cars on the bridge, in this case) in FLTL, can become tricky and complicated.

To address these problems, we propose to introduce the concept of \text{counting fluent}. Suppose that we have the possibility of defining numerical values, that enumerate event occurrences. For instance, \text{CARS\_ON\_BRIDGE} may be a numerical value that keeps count of the number of cars (red or blue) on the bridge. This value is initially 0, is \text{incremented} at each occurrence of an \text{enter} event, and is \text{decremented} at each occurrence of an \text{exit} event. Using \text{CARS\_ON\_BRIDGE}, we can express the weight safety property of the bridge in a more natural way, as follows:

\[
\text{CAPACITY\_SAFE} = \neg (\text{CARS\_ON\_BRIDGE} < \text{CAPACITY} + 1)
\]

Now let us go back to the \text{ONEWAY} property. Assuming the definition of numerical values \text{RED\_CARS\_ON\_BRIDGE} and \text{BLUE\_CARS\_ON\_BRIDGE}, that keep count of the red and blue cars on the bridge, respectively, this property can be specified as follows:

\[
\neg (\text{RED\_CARS\_ON\_BRIDGE} > 0 \lor \text{BLUE\_CARS\_ON\_BRIDGE} > 0)
\]

Our motivating example illustrates two issues. First, it shows that situations in which “counting” events is useful are common. Second, although some properties related to the number of times certain events occur (or are allowed to occur) may be expressed in LTL or FLTL, their specification can be cumbersome. The reader familiar with the formalisms used in this section may be aware that, in some cases, one can simplify the specification of a property by introducing in the model some property related elements (e.g., events that are only enabled when a safety property is violated), and resorting to these elements in the expression of the property. This is a common workaround that, we believe, should be avoided whenever possible, since it mixes the actual model with property related elements, making it harder to understand, and is less declarative, i.e., reasoning about the property’s meaning requires dealing both with an operational part (that incorporated in the model) and a declarative part (that expressed in the logic).

As we will discuss later on, incorporating counting fluents is not a mere syntactic sugar on fluent linear temporal logic. In fact, the resulting logic is strictly more expressive than FLTL. Its associated advantages are to ease the specification of properties that involve counting events in some way (as we have shown in this section), even enabling us to express some properties not expressible in FLTL; and allowing for a cleaner separation of concerns between models and properties, as we will discuss in Section 6. This has a potentially positive impact on understandability, especially taking into account some modern approaches to system description that involve operational component specifications, and constraints on their concurrent interactions.
4. COUNTING FLUENT LTL

To describe more naturally properties of reactive systems in which
enumerating the occurrences of certain events is relevant, we introduce
Counting fluent temporal logic (CFLTL), an extension of fluent linear
temporal logic [11], which complements the notion of fluent by the related concept of
-counting fluent-. Similarly to fluents, counting fluents represent
abstract states in event-based systems whose values depend
on the execution of events. But, as opposed to fluents, which
are logical propositions, counting fluents are numerical val-
ues associated with event occurrences.

Formally, a counting fluent cFl is a 4-tuple defined by
three sets (pairwise disjoint) of events and an initial
numerical value, as follows:

\[ cFl \equiv (I, D, R) \] initially \( N \)

Set \( I \) is the incrementing set of cFl, i.e., when an event of
this set is executed, the value of cFl is incremented by one.
On the other hand, \( D \) represents the decrementing set of
CFLTL, and in this case the value of cFl is decremented when
one of these events occurs. Finally, \( R \) is the resetting events
set, whose execution changes the value of cFl to its initial
value \( N \).

Counting expressions are logical expressions that relate
counting fluents, necessary to deal with their numerical na-
ture. They can be combined with logical and temporal op-
\ers, necessary to deal with their numerical na-
ture. They can be combined with logical and temporal op-
\ers, necessary to deal with their numerical na-

\[ \phi := cFl \sim c | cFl \sim cFl | cFl \sim cFl \pm c \]

s.t., \( c \in N \) and \( \sim \in \{ =, >, < \} \).

Expressions that involve just one counting fluent are called
unary expressions, while the others are called binary expres-
sions. Notice that counting expressions are boolean valued,
they predicate on the values of counting fluents at some
point. Thus, counting expressions can be used as base cases
for formulas. We define the set of well-formed CFLTL for-
\mbox{m}ulas as follows:

(1) every counting expression \( \phi \) is a CFLTL formula;
(2) every propositional fluent \( f \) is a CFLTL formula; and
(3) if \( \varphi_1 \) and \( \varphi_2 \) are CFLTL formulas, then so are \( \neg \varphi_1 \), \( \varphi_1 \lor \varphi_2 \), \( \varphi_1 \land \varphi_2 \), \( \varphi_1[f \lor \varphi_2] \), and the usual derived

\[ \varphi_1[f \lor \varphi_2] \]

definitions for \( \Box \) and \( \Diamond \).

In order to interpret CFLTL formulas, first we introduce
an interpretation for counting fluents. Let \( \Psi \) be a set of
counting fluents. An interpretation for \( \Psi \) is an infinite se-
\mbox{quence} over \( N^\Psi \), that for each instant of time, assigns a
value for each counting fluent. Given an infinite trace \( w = a_1, a_2, \ldots \), we define the function \( V_{i,w}(cFl) \), that denotes the
value of the counting fluent \( cFl \in \Psi \) at position \( i \in N \), as
follows:

\[ V_{i,w}(cFl) = \begin{cases} N & \text{if } i = 0 \\ N + (\#_{l \leq i \leq l+1} a_i \in I) - (\#_{l \leq i \leq l} a_i \in D) & \text{if } i > 0 \end{cases} \]

where \( r \) is the maximum \( l \), with \( 0 \leq l \leq i \), such that \( a_l \in R \),
or \( 0 \), if \( \forall l : 0 \leq l \leq i : a_l \notin R \). Function \( V_{i,w} \) assigns to each

fluence cFl its initial value at the beginning of the execution,
and the value at any other instant of time is obtained by
adding to its initial value the number of occurrences (from its
last resetting event occurrence) of its incrementing events,
and subtracting the number of decrementing events. Notice
that, similar to propositional fluents, our counting fluents are
\textit{close} on the left and \textit{open} on the right, since their values
are updated immediately when a relevant event is executed.

We consider the usual FLTL interpretation for proposi-
tional fluents, logical and temporal operators [11]. Then,
to obtain a complete interpretation of CFLTL formulas, we
define the semantics for the counting expressions as follows:

- \( w, i \models cFl \sim c \iff V_{i,w}(cFl) \sim c \)
- \( w, i \models cFl_1 \sim cFl_2 \iff V_{i,w}(cFl_1) \sim V_{i,w}(cFl_2) \)
- \( w, i \models cFl_1 \sim cFl_2 \pm c \iff V_{i,w}(cFl_1) \pm c \iff V_{i,w}(cFl_2) \pm c \)

where \( c \in N \), \( \sim \in \{ =, >, < \} \) and the symbols \( \sim, + \) and \( - \)
of the right hand side represent the corresponding relation
operation on natural numbers. Notice that the expression
\( cFl_1 \sim cFl_2 \) can be defined as a particular instance of the
expression \( cFl_1 \sim cFl_2 \pm 0 \).

4.1 CFLTL vs. LTL

Let us compare CFLTL and LTL, in terms of expressive-
\mbox{ness} and decidability. It is well known that the expressive
power of LTL is equivalent to that of counter-free Büchi Automata [10].
Intuitively, an automaton is counter-free if it
cannot express, for instance, if a symbol ‘a’ is repeated
\( N \) times in an infinite sequence. CFLTL then results to
be strictly more expressive than LTL, since such “count-
ing” property can straightforwardly be specified in CFLTL,
by using a counting fluent that counts the number of ‘a’s.
Regarding decidability, in [20] it is proven that, if LTL is
extended with diagonal constraints, i.e., expressions of the
form \( \sharp \varphi_1 \sim \sharp \varphi_2 \sim k \), then it becomes undecidable. This
kind of properties are also directly expressible in CFLTL,
turning it into an undecidable logic. In the next section we
develop a sound but incomplete model checking approach
for CFLTL, which shows that our greater expressive power
does not make us fully sacrifice automated analysability.

5. A MODEL CHECKING APPROACH

CFLTL may be suitable to express properties of reactive
systems. However, its adoption would be seriously affected
by the lack of analysis mechanisms for the logic. Model
checking [8] provides an automated method for determining
whether or not a property holds on the system’s state graph,
that is available for FLTL. We study in this section how to
perform model checking of CFLTL properties over systems
described via LTSs, as is the case of FLTL model checking
[11]. At this point, the undecidability of CFLTL leaves us
with two choices. We can search for a decidable fragment
of CFLTL, or we can keep the full expressive power of CFLTL,
and try to define an inherently incomplete (due to the logic’s
undecidability) model checking mechanism for the logic. We
follow the latter in this section.

In order to be able to define a model checking procedure, it
is important to guarantee finiteness of the model and prop-
erties being analysed. Compared to FLTL, our only potential
source of unboundness may come from counting fluents. In
order to keep counting fluents bounded, we propose restrict-
\mbox{ing} them with bounds and scopes, two kinds of numerical
limits to counting fluents, which we describe in detail below. Given the limits to the counting fluents, our approach is based on the definition of a process that monitors the occurrence of the events that update the states of the counting fluents present in the property being analysed. A monitor process activates propositional fluents that capture the truth value of the fluent expressions of the properties formulas, when relevant events occur. Finally, CFLTL formulas are encoded as FLTL formulas, by replacing the counting expressions with corresponding propositional fluents and considering states in which monitors are updating fluent values as unobservable.

The described approach to CFLTL model checking allows us to verify properties containing counting expressions using LTSA [26]. Labelled Transition System Analysar (LTSA) is a verification tool for concurrent systems models. A system in LTSA is modelled as a set of interacting finite state machines. LTSA supports Finite State Process notation (FSP) for concise description of component behaviour, and directly supports FLTL verification by model checking. Syntactically, we propose counting fluents to be defined via the following syntax (extending LTSA’s syntax for propositional fluents):

\[
(CF\text{luentDef}) \ ::= \\
\quad \langle \text{cfluent} \rangle \langle \text{fluent_name} \rangle \langle \text{fluent_bounded} \rangle \quad \langle \text{initial} \rangle \quad \langle \text{limit} \rangle \\
\quad \langle \text{cFluentDef} \rangle \ ::= \\
\quad \langle \text{cfluent} \rangle \langle \text{fluent_name} \rangle \langle \text{fluent_bounded} \rangle \quad \langle \text{initial} \rangle \\
\quad \langle \text{cFluentDef} \rangle \quad \langle \text{limit} \rangle \\
\quad \langle \text{cfluent} \rangle \langle \text{fluent_name} \rangle \quad \langle \text{initial} \rangle \quad \langle \text{limit} \rangle
\]

where brackets and parentheses are used to indicate the kind of limit, bound and scope, respectively, on the corresponding counting fluent.

5.1 Bounds and Scopes

A bound is a limit as parts of arising model, and comes from an actual constraint on the system being specified. For instance, suppose that we are modelling a mobile phone whose volume is restricted to be at most max. Relating this value to events, clearly once max is reached, further presses on the “increase volume” button have no effect on the volume, and therefore can be ignored (at least regarding what concerns the behaviour of the mobile phone). A counting fluent associated with increasing the volume can then be restricted by max as its largest possible value.

Unbounded counting fluents, on the other hand, must be limited by scopes, to maintain the analysis being fully automated. As an example of an unbounded counting fluent, that will have to be limited by a scope, consider an ACK in a model of a TCP protocol (see the example presented in Section 6). As opposed to the case of bounds, which are part of the model, scopes are necessary due to analysis reasons.

When a lower (resp. upper) bound is reached, decrementing (resp. incrementing) events are ignored, i.e., the value of the counting fluent remains the same. When a lower (resp. upper) scope is reached, analysis becomes inconclusive. That is, exceeding a scope during analysis corresponds to reaching fluent overflow states, and thus from models with reachable “overflowed” states nothing can be inferred, neither the validity of the property, nor the construction of a counterexample.

5.2 Model Checking

Let Sys and φ be a FSP specification of a system and a CFLTL property, respectively, and suppose that φ contains fluent expressions. In order to perform the verification process using LTSA, our approach generates a new FSP process Sys′ and a FLTL formula φ′, such thatSys′ incorporates the monitor process that updates the values of the counting fluents and φ′ encodes the propositional fluents associated to each counting expression. The construction of Sys′ and φ′ ensures that every counterexample for φ′ in Sys′ is a counterexample for φ in Sys. Formally, Sys′ ̸\models FLTL φ′ ⇒ Sys ̸\models CFLTL φ.

Below, we describe our approach, consisting of constructing the monitor and the encoded formula φ′.

5.2.1 Monitors for Counting Fluents

Intuitively, a monitor keeps track of the values of the counting fluents (within its bounds/scopes) that appear in a counting expression. For instance, in the case of a unary expression cFlI ∼ c, the monitor records the value of counting fluent cFlI.

Sys′ is obtained by the parallel composition of the system Sys, the monitor process CFMon and a synchroniser process SYNCH. SYNCH is a scheduler process that avoids the interleaving between the events of the system and the updating monitor events, as depicted in the Fig. 1.

The specification of SYNCH is shown in Fig. 2, where Evs is the set of all system events, MonEvs is the set of all event of Sys which are monitored, CFEv is the set of updating events of the CFMon process, and ok is an event of the monitor that indicates that the updating process has been completed.

![Figure 1: Behavioural view of Sys′.](image)

![Figure 2: FSP spec. for SYNCH.](image)
per) limit, the process case has one of the two possibilities depending on the kind of limit. If the limit is a scope, we trigger the fluent overflow event; otherwise, i.e., the limit is a bound, we simply maintain the expression value on its lower (upper) bound.

Note that for every event of the original system considered, the monitor process has cases whose condition guards’ disjunction is always true, i.e., we consider all possibilities for them. This situation, and extending the alphabet's with the rest of events not considered for fluent value update, ensures that the process is non-blocking with respect to the original system behaviour Sys.

5.2.2 Encoding CFLTL formulas

Due to unbounded counting fluents, CFLTL model checking may return one of the following answers: (i) false, when a counterexample is found within the provided scopes, (ii) true, when the property has been proven to hold within the scopes and no fluent overflow was reached, and (iii) maybe, no counterexample was found within the scopes and a fluent overflow state was reached.

In order to verify an CFLTL formula ϕ using LTSA, we encode it as an FLTL formula ϕ’ which captures the truth values of the counting fluent expressions with propositional fluents. Thus, for each counting fluent expression we define a propositional fluent which is activated by the event (update value) of the monitor that satisfies the expression. As an example, if an expression ϵ has the form $cFl \sim e$, then its corresponding propositional fluent is defined by: $\equiv \langle e \bowtie c \rangle, e = c \rangle$.

Notice that there exist some states in Sys’ in which ϕ’ must not be evaluated, namely, when the monitor is updating the counting expression values or a fluent overflow state has been reached. To avoid the analysis on these states, we define the notion of observable states as those that satisfy $OBS \equiv \text{OK} \land \neg F_{\text{overflow}}$, where the fluent $F_{\text{overflow}}$ indicates that a counting fluent has been overflowed. With this notion, the last step of the construction of ϕ’ is based on the translation introduced in [23, Subsection 5.3] to guarantee the exclusion of the non-observable states in the analysis of the validity of ϕ in a model. For instance, if ϕ = □φ, then ϕ’ = □(OBS → φ).

In order to illustrate our model checking process, let us consider the specification of the SLB problem presented in Section 3, and the SAFE_CAPACITY property to be verified. We define the counting fluent $\text{CARS\_ON\_BRIDGE}$ as follows:

\[
\text{cfluent CARS\_ON\_BRIDGE} = \\
\text{< { red[ID].enter, blue[ID].enter }, \{ red[ID].exit, blue[ID].exit }, () > initially 0}
\]

where C is the constant representing the capacity of the bridge. The monitor process generated for the formula is the following:

\[
\begin{align*}
\text{C\text{mon}} &= \text{C\text{mon} B}[0] \\
\text{C\text{mon} B}[1\ldots C+2] &= \\
&\{ \text{when } (i < C+2) \{ \text{red[ID].enter, blue[ID].enter } \rightarrow \text{C\text{mon} B}[i+1] \} \\
&\{ \text{when } (i > C+2) \{ \text{red[ID].enter, blue[ID].enter } \rightarrow \text{flu}nt_{\text{overflow}} \rightarrow \text{ok } \rightarrow \text{C\text{mon} B}[C+2] \} \\
&\{ \text{when } (i > 0) \{ \text{red[ID].exit, blue[ID].exit } \rightarrow \text{cars\text{onBridge}[i-1] } \rightarrow \text{ok } \rightarrow \text{C\text{mon} B}[i-1] \} \\
&\{ \text{when } (i < 0) \{ \text{red[ID].exit, blue[ID].exit } \rightarrow \text{cars\text{onBridge}[i] } \rightarrow \text{ok } \rightarrow \text{C\text{mon} B}[0] \} \\
\end{align*}
\]

Finally, the encoded formula (C instantiated with value 2) and the propositional fluents capturing the values of the corresponding counting expression, are the following:

\[
\begin{align*}
\text{fluent CARS\_ON\_BRIDGE} &= \{ \text{cars\text{onBridge}[0..2],} \\
&\text{cars\text{onBridge}[3..4]} \} \text{ initially True} \\
\text{fluent OK} &= \{ \text{ok, MonEvs } \} \text{ initially True} \\
\text{assert SAFE\_CAPACITY} &= \{ (\text{OK } \& \text{f\_overflow}) \rightarrow \text{CARS\_ON\_BRIDGE} \}
\end{align*}
\]

5.2.3 Verification

Suppose that the encoded formula ϕ’ was successfully verified over system Sys’, i.e., no counterexample was found within the user provided limits for counting fluents. Then, our approach proceeds to check if Sys’ can reach an overflowed state, analysing the formula $\square (\neg F_{\text{overflow}})$. If it is verified over Sys’, i.e., the event fluent_overflow is never executed, then the scopes are big enough to cover the whole state space of the system, so no counterexample of ϕ’ exists. That is, our approach guarantees in this case the validity of property φ in Sys, and returns yes to the verification problem. On the other hand, if an overflowed state is reached, our approach answers maybe indicating that no counterexamples were found in the state space explored, but such space is not the whole state space of the system (a fluent overflow is reachable). This situation may be solved by increasing the scope.

Our model checking approach is supported by the following lemmas. Given a set of events $Evs$ and $A \subseteq Evs$, let us denote by $\mid_A$ the reduction function such that, given a trace $\sigma$ over $Evs$, $\sigma \mid_A$ returns the trace obtained by ignoring the occurrences of events $e \in A$ in $\sigma$. Moreover, let $\Gamma_{Sys}$ and $\Gamma_{Sys’}$ be the sets of execution traces of the LTS of processes $Sys$ and $Sys’$, respectively. Consider that set $C\text{Fset} = C\text{F} Evs \cup \{ \text{ok} \} \cup \{ \text{flu}nt_{\text{overflow}} \}$ contains all events performed by the monitor $CFmon$. Then, the following lemmas hold.

**Lemma 5.1.** For every $\sigma \in \Gamma_{Sys}$, there is a $\sigma’ \in \Gamma_{Sys’}$, such that $\sigma = \sigma’ \mid_{C\text{Fset}}$.

**Lemma 5.2.** Let $\sigma’ \in \Gamma_{Sys’}$ and $\phi$ be a counting fluent expression. Then, for every position $i: \sigma’, i \models (OBS \land F_{\phi}) \Rightarrow \sigma’, i \models F_{\phi}$.

Lemma 5.2 expresses that, within the bounded situations described earlier in this section, the truth value of a counting fluent expression is captured by the corresponding propositional fluent generated over the monitored system. Due to space restrictions, these lemmas’ proofs are not reported here. For a more detailed description of the model checking process, we refer the reader to the technical report available at http://dc.exa.unrc.edu.ar/staff/rdegiovanni/FSE2014.

6. VALIDATION

In this section we validate our proposal, in various aspects. First, we show that counting fluents are well suited to formalise properties in which the occurrences of certain events must be measured. We provide examples showing that the ability of expressing properties that count occurrences of actions allows for a cleaner separation between the behavioural models and the required properties, that otherwise would be tangled together to support verification. Second, we show the relevance of CFLTL by presenting several examples from...
the software engineering literature where the need of counting events arises and is addressed, in our opinion, unsatisfactory. To this end, we have modelled several case studies from a wide range of problem domains where CFTLT allows us to compactly specify complex properties that had previously been defined either informally, or formally but with large and complex FTLT/LTL formulas. Third, we evaluate the quality of CFTLT specifications with respect to two metrics, succinctness [14] and modifiability [4]. To assess succinctness we consider properties taken from the case studies and evaluate how concise CFTLT formulas are with respect to their original FTLT counterparts. The results of this comparison are summarised in Subsection 6.1. Modifiability is tested by proposing reasonable changes to the original requirements for the case studies and evaluating the complexity of introducing them into the original specification, and the one developed by us in CFTLT.

All the assessed case studies are available in the technical report previously referenced. We provide an FSP specification for each model, an informal description and the CFTLT formula of the property to be analysed, accompanied by the monitor generated following our model checking approach, introduced in Section 5.

**Elevator**. In [12], the following informal requirement for an elevator controller is mentioned: “Between the time an elevator is called at a floor and the time it opens its doors at that floor, the elevator can arrive at that floor at most twice.” This property was formalised using LTL, as follows:

```
NoIgnoreTwice = ∃(call ∧ open) →
  \left(\neg\atfloor ∧ \neg\open \land (\open ∨ (\atfloor ∧ \neg\open) \land (\open ∨ ((\atfloor ∧ \neg\open) \land (\open ∨ (\neg\atfloor ∧ \open))))))\right)
```

Dwyer et al. argued that is really difficult to convince oneself that this property captures exactly what is wanted [12]. In order to capture this property, we define the counting fluent `ATFLOOR`, that counts the occurrences of the `atfloor` signal, after a user called the elevator, and specify the property through a CFTLT formula:

```
ATFLOOR ≡ ((atfloor), [], {call}) initially 0
```

Notice that a potential weakening of the requirement would be increasing the number of times the elevator may visit the floor without opening its doors. Such a change would produce a minor modification to the CFTLT formula, while the corresponding FTLT formula would require nesting more temporal and logical operators.

**Timed Light**. Let us consider other simple example, a Timed Light system presented in [23], where a light turns off automatically after 3 time units. Usually in discrete-time event-based systems, the progress of time is modelled with a tick event. Then, using a counting fluent, we are able to capture some interesting timed properties for this system:

```
\text{LightOn} ≡ (\text{on}, \text{off})
\text{RemainsOn} = \square(\text{on} \rightarrow \square(T ≤ 3 \rightarrow \text{LightOn})
\text{EventuallyOffOrPush} = \square(\text{on} \rightarrow (T \leq 3 \land (\text{off} ∨ \text{push})))
```

Intuitively, counting fluent `T` counts the occurrences of `tick`, after the light is turned on. Property `RemainsOn` requires that once the light has been turned on, it must remain on during the next 3 time units. Property `EventuallyOffOrPush` expresses that when the light is turned on, it will be eventually turned off within 3 time units, except if a push event occurs during that time. These timed properties can be specified in FTLT as follows:

```
\text{RemainsOn} = \square(\text{on} \rightarrow (\text{LightOn} ∧ \square(\neg\text{tick} \land \text{LightOn} ∧
  \square(\neg\text{tick} \land \text{LightOn} ∧
  \square(\neg\text{tick} \land \text{LightOn}))))))
```

**ATM.** Consider the model of an ATM machine depicted in Figure 3, taken from [34].

```
\text{LOCKED} ≡ (\text{lockAcct}, \text{unlockAcct})
\text{ERRORS} ≡ ((\text{badPwd}), [], \{\text{correctPwd, unlockAcct}\}) initially 0
\text{WrongExtraction} = \square(\neg(\text{ERRORS} \geq 3 ∧ \text{money})
\text{IncorrectLock} = \square(\neg(\text{ERRORS} < 3 ∧ \text{LOCKED})
```

Figure 3: LTS that models the ATM’s behaviour.

Initially, the ATM requests the user to insert a card and enter the password (Pwd). The validity of the password is verified, and if its verification succeeds, the user can extract money and remove the card from the ATM. Otherwise, when the password is incorrect, the ATM system starts “counting” the number of successive mistakes made by the user. In case the user makes three consecutive mistakes, the ATM locks the account, preventing money extractions. Notice that the user is able to remove its card form the ATM, provided that his account is not locked.

A typical security mechanism that ATMs implement consists of blocking account access when the user makes three consecutive mistakes at entering his password. In order to check whether the ATM correctly implements this security mechanism, we may specify the following fluent definitions and CFTLT properties:
Intuitively, property *WrongExtraction* expresses that it cannot be the case that, after three consecutive wrong password insertions, the user extracts his money. Property *IncorrectLock* expresses that the account cannot be locked when less than three incorrect password insertions have happened. Let us see now how can we specify these properties in FLTL.

\[ \text{BadPass} \equiv \{(\text{badPwd}) \}, \{\text{correctPwd.unlockAcct}\} \]

\[ \text{WrongExtraction} = \]

\[ \Diamond \neg \text{badPwd} \land \Box \text{BadPass \land} \Box \text{BadPass \land} \]

\[ \Box \text{BadPass \land} \Box \text{BadPass \land} \text{MAX}\text{Window} \]\\

\[ \text{IncorrectLock} = \Diamond \neg(\text{BadPass \land LOCKED}) \land \]

\[ \neg \text{BadPass \land} \Box \neg \text{BadPwd \land} \left( \text{BadPwd} \land \Box \neg \text{BadPwd \land} \text{MAX}\text{Window} \right) \]\\

Notice that, despite FLTL is expressive enough to specify these properties, the encoding is not straightforward. The main problem lies in the complex nesting of logical and temporal operators, needed to represent the number of times that badPwd was executed.

Suppose the bank policy changes, and now it tolerates only two successive wrong password insertions. Clearly, this kind of changes in the CFLTL formulas can be easily applied changing the constant 3 for 2, while in the FLTL formulas we have to modify the temporal operator nesting.

**TCP Sliding Window.** Consider the TCP network protocol [33], a reliable delivery service that guarantees that all packages received will be identical with packages sent. Figure 4 shows a LTS that models the behaviour of a single package along a TCP network communication.

![Figure 4: Specification of TCP package behaviour.](image)

Initially, the package that encapsulates the sender’s information is waiting to be sent. Once the package is sent, two situations have to be considered: the package can be lost in the communication, or it can be received by the receiver. When the sending fails, a timeout waiting for the acknowledgment will occur, so then the sender resends the package to avoid the loss of packages. In the case of a successful reception, the receiver processes the package and returns an ack to the sender. If the ack message gets lost, a timeout occurs in the sender and it resends the (not acknowledged) package. In this situation, the receiver will discard the duplicated package. We represent the traffic in a network, combining the behaviour of various packages (PACKs).

The TCP protocol has been improved many times, to optimize network transfer. In particular, the **TCP Sliding Window** protocol modifies dynamically the size of the window depending on the channel reliability. The term window refers to the number of packages that can be sent without receiving their corresponding acks. This protocol considers that initially the window size is 1 and each time that an ack is received, the window size is incremented by one (i.e., the channel’s reliability is increased), until the maximum size for the window is reached (in our case, assume MAX = 5).

When any loss in the channel is detected, i.e., when a timeout occurs, the channel becomes less reliable, so then the window size is decremented by 1 (when it is greater than 1).

Over this protocol specification, an interesting property to check is that the sender can be waiting at most for MAX acks. Using CFLTL we can specify this property as follows:

\[ \text{ACK} \equiv \{\text{PACKs.ack}\}, \{\text{PACKs.timeout}\} \] \[ \text{ACK}_{\text{less-MAX}} = \Box (\text{ACK} \leq \text{MAX}) \]

Notice that the above property is weaker than what we would actually need to express. Indeed, the property expresses that the number of packages not acknowledged is bounded by the maximum window size, whereas we would like to express a stronger property, saying that the number of packages not acknowledged is bounded by the current window size. This stronger property needs to refer to a dynamic value, a value that changes as the system is executing.

So let us define \( \text{WINDOW} \), a counting fluent that maintains the current window size (i.e., it is incremented when an ack is received and decremented when a timeout occurs).

Now, we can easily specify this dynamic property as follows:

\[ \text{WINDOW} \equiv \{\text{PACKs.ack}\}, \{\text{PACKs.timeout}\} \] \[ \text{ACK}_{\text{less-WINDOW}} = \Box (\text{ACK} \leq \text{WINDOW}) \]

In contrast to the previous examples, we cannot express the same properties in FLTL nesting temporal and logical operators. Consider the following formula:

\[ \text{SixSends.NoAck} = \Diamond \neg\{\text{PACKs.send} \land \Box \neg\{\text{PACKs.ack} \land \}

\[ \text{PACKs.send} \land \Box \neg\{\text{PACKs.ack} \land \}

\[ \text{PACKs.send} \land \Box \neg\{\text{PACKs.ack} \land \}

\[ \text{packs.send} \land \Box \neg\{\text{packs.ack} \land \text{packs.send})\] \]

This formula indicates that it cannot be the case that six successive send events occur without receiving an ack. However, this is just a particular case of \( \text{ACK}_{\text{less-MAX}} \). It does not consider, for instance, the case when 4 successive send occur, then an ack takes place, and finally 3 more ack (sending 6 packages without receiving acknowledgments) occur. A similar problem was described in Section 3. For similar reasons, the dynamic property \( \text{ACK}_{\text{less-WINDOW}} \) cannot be expressed in FLTL, using the same strategy followed in Section 3.

Our proposed FLTL solution to capture this property involves introducing \( N \) fluents, one per package (PACKs=[1..N]), to indicate if the package has or has not been acknowledged. Then, the FLTL formula should encode the possibilities in which more than \( \text{MAX}=5 \) sent packages have not been acknowledged.

\[ \text{ACK}_{p:1..N} = \{\{\text{pack.send}\}, \{\text{pack.ack}\} \]

\[ \text{ACK}_{\text{less-MAX}} = \]

\[ \Diamond \neg\{\text{ACK}[1..6] \lor \text{ACK}[2..7] \lor \ldots \lor \text{ACK}[N-6..N] \] 

\[ \lor (\text{ACK}[1] \land \text{ACK}[3..7]) \lor (\text{ACK}[1] \land \text{ACK}[4..8]) \ldots ) \]
Notice that we have \( \binom{n}{3} \) cases in which property \( \text{ACK} \leq \text{MAX} \) is violated. Thus, for instance, for 10 packages, the FLTL formula will contain 210 disjunctions. Clearly, capturing this “counting property” in FLTL is infeasible to do manually, definitely error prone.

Notice that any change to the protocol may produce a big change in the encoded FLTL formula. On the other hand, consider for example the following modification to the protocol: when a timeout is detected, the channel is considered unreliable, so the window size should be resetted to one. In this case, the CFLTL formulas remains as they are, but we need to modify the definition of the counting fluent \( \text{WINDOW} \) (now the timeout event resets its value, instead of decrementing it).

**Producers and Consumers.** This is a classic concurrency problem, where there are producers (PROD), consumers (CONS) and a buffer of capacity C. The producers put values into the buffer and the consumers get them from it. The usual solutions to this problem consider two semaphores to synchronise the processes of producing and consuming. More precisely, a semaphore \( \text{full} \) models the number of values in the buffer (initially 0); and a semaphore \( \text{empty} \) indicates the free space in the buffer (initially C). A semaphore \( S \) is a numeric variable equipped with two operations, historically denoted as \( V \) (increments \( S \)) and \( P \) (decrements \( S \)). An FSP specification is available in [26]. To check if producers and consumers implement correctly the synchronisation protocol, we can use counting fluents to specify the following properties:

\[
\begin{align*}
V_{\text{prod}} & \equiv \{\{\text{empty}.P\}, \{\text{PROD}.\text{put}\}\}, \{\}\text{ initially 0} \\
V_{\text{cons}} & \equiv \{\{\text{full}.P\}, \{\text{CONS}.\text{get}\}\}, \{\}\text{ initially 0} \\
\text{CorrectProduction} & = \Box(V_{\text{prod}} \geq 0) \\
\text{CorrectConsumption} & = \Box(V_{\text{cons}} \geq 0)
\end{align*}
\]

Intuitively, property \( \text{CorrectProduction} \) checks if the producers test if there is space in the buffer before producing, and property \( \text{CorrectConsumption} \) checks if the consumers verify if the buffer is not empty before consuming. Moreover, we can specify a more general property that relates the current values of each semaphore (i.e., a dynamic property) as follows:

\[
\begin{align*}
\text{CountP} & \equiv \{\{\text{empty}.V\}, \{\text{empty}.P\}\}, \{\}\text{ initially C} \\
\text{CountC} & \equiv \{\{\text{full}.V\}, \{\text{full}.P\}\}, \{\}\text{ initially 0} \\
\text{CorrectSynchronisation} & = \Box(\text{CountP} + \text{CountC} \leq \text{C})
\end{align*}
\]

\( \text{CorrectSynchronisation} \) expresses that the number of occupied spaces plus the number of free spaces \( (\text{CountP} + \text{CountC}) \) in the buffer should be always less than the capacity \( C \). It may not be exactly equal to \( C \), because the semaphore’s values are incremented (executing \( V \)) only after effectively producing/consuming the value from the buffer.

In contrast to the TCP example, here the events over the semaphores are shared by the producers and consumers for synchronisation, i.e., we do not distinguish which producer or consumer executes a \( P \) or \( V \) over a semaphore. Then, both techniques we proposed to specify counting properties in FLTL do not apply in this case. Inevitably, to overcome this problem, we have to modify the model.

### 6.1 Succinctness Evaluation

Succinctness [14] is a common measure used for comparing how short two logics can specify certain properties. Usually, this measure is applied to logics with the same expressive power. However, it may result interesting to know if those properties that can be expressed in both logics can be expressed more succinctly in one of the logics.

We summarise in Table 1 the comparison between the CFLTL and FLTL formulas specified for the different case studies along the paper. More precisely, the attributes we consider are: (F) the number of counting/propositional fluents and events involved in the formulas; (N) the maximum nesting level of temporal operators; the number of temporal operators (TO) and logical operators (LO) used in the formulas. We report these values in the table in the following order (F) N/TO/LO.

<table>
<thead>
<tr>
<th>Example</th>
<th>Property</th>
<th>CFLTL</th>
<th>FLTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited Capacity Bridge</td>
<td>CAPACITY SAFE</td>
<td>(4)</td>
<td>(8)</td>
</tr>
<tr>
<td>Elevator</td>
<td>No Ignore Fence</td>
<td>(3)</td>
<td>(8)</td>
</tr>
<tr>
<td>Timed Light</td>
<td>Remains On</td>
<td>(3)</td>
<td>(4)</td>
</tr>
<tr>
<td>ATM</td>
<td>Wrong Extraction</td>
<td>(2)</td>
<td>(3)</td>
</tr>
<tr>
<td></td>
<td>Incorrect Lock</td>
<td>(2)</td>
<td>(3)</td>
</tr>
<tr>
<td>TCP</td>
<td>ACK less MAX</td>
<td>(1)</td>
<td>(10)</td>
</tr>
<tr>
<td></td>
<td>ACK less WINDOW</td>
<td>(2)</td>
<td></td>
</tr>
<tr>
<td>Prod-Cons</td>
<td>Correct Production</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Correct Consumption</td>
<td>(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CorrectSynchronisation</td>
<td>(2)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Succinctness Evaluation for CFLTL.

The first result that can be observed from the table is the considerable difference in the number of fluents used in some formulas. For instance, when specifying FLTL properties like \( \text{CAPACITY SAFE} \) and \( \text{ACK less MAX} \), we need to identify each car or TCP package (with a fluent), to count how many cars are on the bridge or the number of sent packages that have not been acknowledged yet, respectively. Then, we encode the whole set of possibilities in which the number of cars exceeds the bridge capacity or the number of not acknowledged packages is greater than the window size. This leads to a FLTL formula that contains a rather large number of logical operators (e.g., 210 disjunctions for 10 TCP packages).

On the other hand, when we specify particular sequences of events in FLTL (e.g., three successive wrong password insertions in the ATM example), we tend to nest temporal operators to count the occurrences of these events. However, when this nesting becomes complex, it is really difficult to convince oneself that the property captures what one wants. Moreover, we showed that changes cannot straightforwardly be applied, and we should carefully introduce the new operators nesting. With respect to the reasonable modifications we proposed, the CFLTL formulas did not suffer major changes. In some cases the CFLTL formula remained the same, but what changed were the counting fluent definitions (e.g., the \( \text{WINDOW} \) counting fluent definition).

An interesting point to highlight is that we did not have to instrument the model to support our verification purposes. As Table 1 shows, our CFLTL formulas require a few fluent definitions and are naturally expressed, without the need of nesting a considerable number of temporal and logical operators.
Finally, it is worth to mention that the concept of counting fluents was effectively useful to capture dynamic properties over the systems we studied.

6.2 Further Examples

In addition to the examples presented, here we report a set of properties found in the literature where the need of counting particular events is present. Due to space restrictions, we only report the informal description of the properties and the reference from which it was extracted:

- [5]: (1) count the number of times the Store reports that a requested item is unavailable; (2) count the number of items offered to the User before the User accepts to buy; (3) count the total number of times the Bank has refused the credentials of the Store.
- [24]: For example, to control the overall system performance, the auction service provider may want to set a limit on the number of bids that a bidder can make during each session: “Bid exists at most 3 times after Login until Logout”.
- [6]: “If a certain patient presses the panic-button three times during a time span of a week, the First-Aid Squad must hospitalize the patient within one day”.
- [17]: (1) The customer should pay the selected offer by providing his credit card data within 30 minutes after the reservation step. Otherwise, the reserved offer will be canceled; (2) In a 5 minutes interval the customer can only do 3 failed payment trials; (3) The customer can cancel a travel reservation the latest 7 days before his travel; (4) The customer can change his travel reservation only 2 times. Changes are only allowed between 1 day and 5 days after the reservation date.

7. RELATED WORK AND CONCLUSIONS

Classical Temporal Logics such as LTL and CTL are convenient formalisms for specifying reactive systems and their properties. Several quantitative extensions of these formalisms have been studied, such as timed and probabilistic temporal logics [2, 3]. Among them, Counting CTL [21] and Counting LTL [20] are extensions of CTL and LTL that allow one to express constraints over the number of times that certain sub-formulas are satisfied along a run. Unlike our approach, Counting LTL cannot predicate over the relation between the number of occurrences of relevant events. In [7], Bianculli et al. propose SOLOIST, a formalism based on many-sorted first-order metric temporal logics, more specifically Metric Linear Temporal Logic with Past (MPLTL) [31]. SOLOIST provides support for some aggregate operators for events occurring in a certain time window. The main advantage of SOLOIST is that it translates to MPLTL which reduces to PLTL [18], hence, it can be analysed and verified by a wide number of techniques and tools. However, such an advantage also imposes limitations in expressiveness. For instance, it is impossible to compare among occurrences of events, hence, a simple example such as the Single Lane Bridge could not be described as properties relating occurrences of blue and red cars could not be defined (e.g. $∃(\text{blue cars}) \geq ∃(\text{red cars})$).

In the AI planning [13, 22] community a technique to capture numeric values with fluents has been proposed, namely additive fluents [22]. Additive fluents provide support for describing measurable quantities, such as money and memory. Typically, they are incremented/decremented by the execution of actions, e.g., allocating/deallocating memory. In contrast to our approach, additive fluents are used explicitly in the specification, i.e., they can be mentioned in operations preconditions and postconditions. Also, each operation describes how additive fluents’ values are updated.

As mentioned in Section 1, the approach introduced in this article is more closely related to fluent temporal logic (FLTL) [11]. FLTL enables specifying LTL properties of event-based systems, where propositions are capture via fluents. Fluents represent state propositions, defined in terms of activating and deactivating events. As opposed to the binary nature of fluents, in this paper we introduced the related concept of counting fluent. Intuitively, counting fluents are numerical values, that enumerate event occurrences, and allow one to specify naturally some properties of reactive systems, such as bounded liveness and other properties in which the number of occurrences of certain events is relevant. A particular extension to FLTL that is related to our proposal is that presented in [23], in which special temporal operators for modelling timed properties of discrete-time event-based models are introduced (basically, temporal operators are equipped with bounds (e.g., $\bigcirc_{\leq d}$ and $\bigcirc_{< d}$) for counting the progress of time). The work in [23] shows the necessity for counting the occurrences of certain events, although the proposal is limited to counting a very particular event, namely $\text{tick}$, which models the time passing.

Counting fluent temporal logic, our proposed extension of fluent linear temporal logic, provides the flexibility to specify naturally some properties of event-based systems, that require counting events. We proved that the logic is strictly more expressive than LTL, and is also undecidable. However, we accompanied this introduced formalism with a sound but incomplete (due to the undecidability of the logic) model checking approach, that allows us to analyse CFLTL formulas under user defined scopes. Moreover, we performed a strong validation of multiple aspects of our proposal. We showed that counting fluents are well suited to describe properties in which enumerating the occurrences of certain events is relevant, and we exhibited that this kind of cases are very common in the research literature. In addition, we carried out a qualitative evaluation on CFLTL formulas. More precisely, we consider two metrics, succinctness and modifiability, to compare the same properties specified in CFLTL and FLTL. As a result, in our opinion the succinctness of CFLTL properties makes them more understandable and modifiable.

As work in progress, we are improving our prototypical tool that automates the generation of the monitor process and the encoding of CFLTL formulas into FLTL. In addition, we are working on the formal underpinnings of the introduced logic, studying the details of the undecidability of CFLTL and the complexity that the monitor process may reach (when we use scopes for model checking purposes). As future work, we plan to study which fragments of CFLTL are decidable, and subject to sound and complete model checking procedures.
REFERENCES


