Specifying Event-Based Systems with a Counting Fluent Temporal Logic

Germán Regis*, Renzo Degiovanni*‡, Nicolas D'Ippolito†‡, Nazareno Aguirre*‡

*Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Argentina
†Departamento de Computación, FCEyN, Universidad de Buenos Aires, Argentina
‡Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

Abstract—Fluent linear temporal logic is a formalism for specifying properties of event-based systems, based on propositions called *fluents*, defined in terms of activating and deactivating events. In this paper, we propose complementing the notion of fluent by the related concept of *counting fluent*. As opposed to the boolean nature of fluents, counting fluents are numerical values, that enumerate event occurrences, and allow us to specify naturally some properties of reactive systems.

Although by extending fluent linear temporal logic with counting fluents we obtain an undecidable, strictly more expressive formalism, we develop a sound (but incomplete) model checking approach for the logic, that reduces to traditional temporal logic model checking, and allows us to automatically analyse properties involving counting fluents, on finite event-based systems.

Our experiments, based on relevant models taken from the literature, show that: *(i)* counting fluent temporal logic is better suited than traditional temporal logic for expressing properties in which the number of occurrences of certain events is relevant, and *(ii)* our model checking approach on counting fluent specifications is more efficient and scales better than model checking equivalent fluent temporal logic specifications.

I. 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 [20], [10], 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 [16], CCS [31] 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 [29], [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 [14] 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 boolean 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 nestings of temporal operators in linear temporal logic, while we conveniently capture them using counting fluents.

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. We show that although the introduced logic is an undecidable, strictly

more expressive, extension of fluent linear time temporal logic, we can 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, that is implemented in a tool extending LTSA [19], the above described convenience of our introduced formalism for property specification is achieved without having to fully sacrifice automated analysability.

In order to evaluate our proposed formalism and model checking approach, we performed various experiments, based on relevant models taken from the literature. Our experiments show that: *(i)* counting fluent temporal logic is better suited than traditional fluent temporal logic for expressing properties in which the number of occurrences of certain events is relevant, and *(ii)* our model checking approach on counting fluent specifications is more efficient and scales better than model checking equivalent fluent temporal logic specifications.

The remainder of this article is organised as follows. Section II introduces preliminary concepts necessary in the paper. Section III 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 IV describes Counting Fluent Temporal Logic in detail. We present the model checking approach in Section V. In Section VI we evaluate our proposed formalism and model checking approach. Finally, we discuss related work in Section VII and present our conclusions in Section VIII.

II. BACKGROUND

Labelled Transition Systems (LTS) are typically used to model the behaviour of interacting components, characterised by states and transitions between them [26]. Formally, a LTS P is a quadruple $\langle Q, A, \delta, q_0 \rangle$, where Q is a finite set of states, A is the *alphabet* of P (a subset of the universe 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 a 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) [26] is a process algebra, whose expressions can be automatically mapped into finite LTS, and vice versa. In FSP specifications, "`->`" denotes event prefix, "`|`" denotes choice, and conditional choices can be expressed by means of "`when`" clauses. Processes may be indexed and parameterised, and can be composed in a sequential ("`;`") or parallel way ("`||`").

Fluent Linear Time Temporal Logic (FLTL) is a variant of LTL [27], [28], well suited for describing state properties of event-based discrete systems (e.g., LTS) [14]. FLTL extends LTL by incorporating the possibility of describing certain abstract states, called *fluents*, characterised by events of the system. Miller and Shanahan informally define propositional fluents as time-varying properties of the world [30]. Formally,

$Fl \equiv \langle I, T, B \rangle$ defines a fluent Fl, where $I, T \subseteq A$, $I \cap T = \emptyset$, and $B \in \{\textit{true}, \textit{false}\}$. B indicates the initial value of Fl. When any event in I occurs, the fluent starts to be true, and it becomes false again when any event in T occurs. If the term B is omitted then Fl is initially *false*. It is possible to combine fluents and events in FLTL formulas. Specifically, every event e has a fluent Fl_e associated, whose initial set of actions is the singleton $\{e\}$ and whose terminating set is $A \setminus \{e\}$.

A FLTL formula is a LTL formula where propositions are fluents. Given a set Φ of fluents, well-formed FLTL formulas are defined inductively using the standard boolean operators and the temporal operators $\bigcirc$ (next) and $\mathcal{U}$ (strong until), in the following way: *(i)* every $fl \in \Phi$ is a formula, and *(ii)* if ϕ and ψ are formulas, then so are $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$, $\bigcirc\phi$, $\phi\mathcal{U}\psi$. We consider the usual definition for the operators $\square$ (always), $\diamond$ (eventually) and $\mathcal{W}$ (weak until) in terms of $\bigcirc$ and $\mathcal{U}$, and boolean connectives.

III. MOTIVATING EXAMPLE

We will start to motivate our work in this paper with the Single Lane Bridge Problem (SLB), a modelling problem introduced in [26] (cf. Section 7.2 therein). The problem consists of a narrow bridge which only allows for a single lane of traffic, which must be appropriately controlled to avoid safety violations. As one may expect, a safety violation occurs if two cars moving in different directions are on the bridge at the same time. In order to simplify the presentation of the problem, cars moving in different directions are represented by different colours: red and blue cars.

```
const N = 4 // number of each type of car
range T = 0..N // type of car count
range ID = 1..N // car identities
BRIDGE = BRIDGE[0][0], //initially empty
BRIDGE[nr:T][nb:T] = //nr (nb) is the red (blue) counter
    (when (nb==0) red[ID].enter ->BRIDGE[nr+1][nb]
    |            red[ID].exit ->BRIDGE[nr-1][nb]
    |when (nr==0) blue[ID].enter ->BRIDGE[nr][nb+1]
    |            blue[ID].exit ->BRIDGE[nr][nb-1]).
NOPASS1 = C[1], C[i:ID] = ([i].enter ->C[i%N+1]).
NOPASS2 = C[1], C[i:ID] = ([i].exit ->C[i%N+1]).
CAR = (enter->exit->CAR). //car definition
||CONVOY = ([ID]:CAR || NOPASS1 || NOPASS2).
||CARS = (red:CONVOY || blue:CONVOY).
||SingleLaneBridge = (CARS || BRIDGE).
```

In this model, the CAR process specifies a simplified behaviour of a car with respect to the bridge. Process BRIDGE is essentially what controls the access to the bridge: it prevents cars in one direction entering the bridge when cars in the opposite direction are already on the bridge. The NOPASS processes strengthen the model, avoiding cars to pass over on the bridge. Finally, the system is modelled as the composition of the BRIDGE process with the instances of cars specified by means of processes CONVOY and CARS.

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) and define one fluent 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 *RED[i:ID]*=`<red[i].enter, red[i].exit>`. Then, the required safety property is specified as follows:

$$ONEWAY = \Box\neg((RED[1] \lor RED[2] \lor \ldots \lor RED[N]) \land$$
$$(BLUE[1] \lor BLUE[2] \lor \ldots \lor BLUE[N]))$$

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 this kind 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:T][nb:T] = //nr (nb) is the red (blue) counter
 (when((nb==0)&&(nr<C)) red[ID].enter ->BRIDGE[nr+1][nb]
 |                      red[ID].exit ->BRIDGE[nr-1][nb]
 |when((nr==0)&&(nb<C)) blue[ID].enter->BRIDGE[nr][nb+1]
 |                      blue[ID].exit ->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 *ONEWAY* property holds, we can specify the bridge's weight safety as the following property *CAPACITY_SAFE*:

$$\Box\neg((RED[1] \land RED[2] \land RED[3]) \lor (RED[1] \land RED[2] \land RED[4]) \lor$$
$$(RED[1] \land RED[3] \land RED[4]) \lor (RED[2] \land RED[3] \land RED[4]) \lor$$
$$(BLUE[1] \land BLUE[2] \land BLUE[3]) \lor (BLUE[1] \land BLUE[2] \land BLUE[4]) \lor$$
$$(BLUE[1] \land BLUE[3] \land BLUE[4]) \lor (BLUE[2] \land BLUE[3] \land BLUE[4]))$$

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 *counting fluent*. Suppose that we have the possibility of defining numerical values, that enumerate event occurrences. For instance, *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 *incremented* at each occurrence of an `enter` event, and is *decremented* at each occurrence of an `exit` event. Using *CARS_ON_BRIDGE*, we can express the weight safety property of the bridge in a more natural way, as follows:

$$CAPACITY\_SAFE = \Box(CARS\_ON\_BRIDGE < \texttt{capacity} + 1)$$

Let us now go back to the *ONEWAY* property. Assuming the definition of numerical values *RED_CARS_ON_BRIDGE* and *BLUE_CARS_ON_BRIDGE*, that keep count of the red and blue cars on the bridge, respectively, this property can be specified as follows:

$$\Box\neg(RED\_CARS\_ON\_BRIDGE > 0 \land BLUE\_CARS\_ON\_BRIDGE > 0)$$

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 VI. 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.

IV. 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 [14], 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 values 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:

$$cFl \equiv \langle I, D, R \rangle \text{ initially } N$$

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 cFl, 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*.

Counting expressions are logical expressions that relate counting fluents, necessary to deal with their numerical nature. They can be combined with logical and temporal operators to specify CFLTL formulas. For instance, a counting expression can compare the values of two of counting fluents, or query for the value of a particular counting fluent. Formally, given a set Ψ of counting fluents and $cFl_1, cFl_2 \in \Psi$, a valid counting expression ϕ is defined as follows:

$$\phi ::= cFl_1 \sim c \mid cFl_1 \sim cFl_2 \mid cFl_1 \sim cFl_2 \pm c$$

where $c \in \mathbb{N}$ and $\sim \in \{=, >, <\}$. Expressions that involve just one counting fluent are called *unary* expressions, while the others are called *binary* expressions. 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 formulas as follows:

(1) every counting expression ϕ is a CFLTL formula;
(2) every propositional fluent f is a CFLTL formula; and
(3) if φ_1 and φ_2 are CFLTL formulas, then so are $\neg\varphi_1$, $\varphi_1 \vee \psi_2$, $\varphi_1 \wedge \psi_2$, $\bigcirc\varphi_1$, $\varphi_1 \mathcal{U} \varphi_2$, and the usual derived definitions for $\square\varphi_1$, $\diamond\varphi_1$ and $\varphi_1 \mathcal{W} \varphi_2$.

In order to interpret CFLTL formulas, first we introduce an interpretation for counting fluents. Let Ψ be a set of counting fluents. An interpretation for Ψ is an infinite sequence over $\mathbb{N}^\Psi$, 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 $\mathcal{V}_{i,w}(cFl)$, that denotes the value of the counting fluent $cFl \in \Psi$ at position $i \in \mathbb{N}$, as follows:

$$\mathcal{V}_{i,w}(cFl) = \begin{cases} N \text{ if } i = 0 \\ N + (\#_{r \leq k \leq i} a_i \in I) - (\#_{r \leq k \leq i} a_i \in D) \text{ if } i > 0 \end{cases}$$

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 $\mathcal{V}_{i,w}$ assigns to each fluent 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

close on the left and *open* on the right, since their values are updated immediately when a relevant event is executed.

We consider the usual FLTL interpretation for propositional fluents, logical and temporal operators [14]. Then, to obtain a complete interpretation of CFLTL formulas, we define the semantics for the counting expressions as follows:

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

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

A. CFLTL vs. LTL

Let us compare CFLTL and LTL, in terms of expressiveness and decidability. It is well known that the expressive power of LTL is equivalent to that of counter-free Büchi Automata [11]. 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 "counting" property can straightforwardly be specified in CFLTL, by using a counting fluent that counts the number of 'a's. Regarding decidability, in [21] it is proven that, if LTL is extended with *diagonal constraints*, i.e., expressions of the form $\sharp\varphi_1 - \sharp\varphi_2 \sim k$, 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.

V. 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 LTS, as is the case of FLTL model checking [14]. 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 properties being analysed. Compared to FLTL, our only potential source of unboundedness may come from counting fluents. In order to keep counting fluents bounded, we propose restricting 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 Analyser (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):

$\langle CFluentDef \rangle ::= $ **'cfluent '** $\langle fluent\_name \rangle$ $\langle fluent\_bounds \rangle$ **'='**
'<'$\langle incremental\_events\_set \rangle$ **','** $\langle decremental\_events\_set \rangle$ **','**
$\langle reset\_events\_set \rangle$ **'>'** **'initially '** $\langle initial\_value \rangle$
$\langle fluent\_bounds \rangle ::= ($'['' | '('') $\langle min\_value \rangle$'..'$\langle max\_value \rangle$ (']' | ')')

---

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

A. Bounds and Scopes

A *bound* is a limit that arises as part of modelling, 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 VI). 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.

B. 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 that `Sys'` 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'` $\not\models_{FLTL} \phi' \Rightarrow$ `Sys` $\not\models_{CFLTL} \phi$.

Below, we describe our approach, consisting of constructing the monitor and the encoding formula ϕ'.

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 $cFl \sim c$, the monitor records the value of counting fluent cFl.

`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 events of `Sys` which are monitored, `CfEvs` is the set of updating events of the `CFmon` process, and `ko,ok` are events of the monitor indicating that the update process is being carried out.
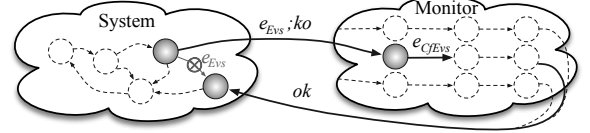

Fig. 1. Behavioural view of `Sys'`.

```
SYNCH = ({Evs\MonEvs} ->SYNCH | MonEvs ->ko ->CFSYNCH),
CFSYNCH = (CfEvs ->CFSYNCH | ok ->SYNCH).
```

Fig. 2. FSP spec. for `SYNCH`.

Basically, the monitor considers the system's events involved in the formula ϕ being verified. The monitor process contains one parameter for each counting fluent appearing in ϕ. The process body is composed of cases (choices) considering monitored events of the system and, as guards, conditions regarding values of the corresponding counting fluents. Each case triggers a sequence of updating events, ending with the `ok` event indicating that the updating process was completed.

The new values for the counting fluents (parameters of the monitor), are calculated in terms of the membership of the event to the incrementing, decrementing or resetting event sets of the corresponding counting fluent definitions.

In case some parameter is in a boundary situation, i.e., when the counting fluent value reaches its lower (upper) limit, the process case has one of 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 maintain the expression value on its lower (upper) bound.

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

The additional cost introduced by the monitoring task in `Sys'`, depends on the formula to be analysed. In terms of state space, it is in the worst case the product between the number of monitored system events `MonEvs` and the size of the monitor process `CFmon`, all possible combinations of the counting fluent values.

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 a CFLTL formula ϕ using LTSA, we encode it as a 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 has the form $cFl \sim c$, then its corresponding propositional fluent is defined by: $\equiv \langle \epsilon[\sim c], \epsilon[\nsim c] \rangle$.

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 $OBS \equiv OK \wedge \neg F\_overflow$, where the fluent `F_overflow` indicates that a counting fluent has overflowed. With this notion, the last step of the construction of ϕ' is based on the translation introduced in [24] to guarantee the exclusion of the non-observable states in the analysis of the validity of ϕ in a model. For instance, if $\phi = \Box\varphi$, then $\phi' = \Box(OBS \to \varphi)$.

In order to illustrate our model checking process, let us consider the specification of the SLB problem presented in Section III, and the *SAFE_CAPACITY* property to be verified. We define the counting fluent *CARS_ON_BRIDGE* as follows:

```
cfluent CARS_ON_BRIDGE [0..C+2]=
    < { 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:

```
CFmon = CFmon_B[0], CFmon_B[i:0..C+2] =
 ( when (i<C+2) {red[ID].enter,blue[ID].enter}
                    -> carsOnBridge[i+1] ->ok ->CFmon_B[i+1]
 | when (i>=C+2) {red[ID].enter,blue[ID].enter}
                    -> fluent_overflow ->ok ->CFmon_B[C+2]
 | when (i>0) {red[ID].exit,blue[ID].exit}
                    -> carsOnBridge[i-1] ->ok ->CFmon_B[i-1]
 | when (i<=0) {red[ID].exit,blue[ID].exit}
                    -> carsOnBridge[i] ->ok ->CFmon_B[0]).
```

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

```
fluent CARS_ON_BRIDGE_LESS_3 = <carsOnBridge[0..2],
                    carsOnBridge[3..4]> initially True
fluent OK =<ok,ko> initially True
assert SAFE_CAPACITY=[]((OK &&!F_overflow)
                            ->CARS_ON_BRIDGE_LESS_3)
```

3) Verification: Suppose that the encoding 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 $\Box(\neg F\_overflow)$. 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 $A \subset Evs$, let us denote by $|_A$ the *reduction* function such that, given a trace σ over Evs, $\sigma \mid_A$ returns the trace obtained by ignoring the occurrences of events $e \in A$ in σ. Moreover, let $\Gamma_{\texttt{Sys}}$ and $\Gamma_{\texttt{Sys'}}$ be the sets of execution traces of the LTS of processes `Sys` and `Sys'`, respectively. Consider that set *CFset* contains all events performed by the monitor `CFmon`. Then, the following lemmas hold.

Lemma 5.1: For every $\sigma \in \Gamma_{\texttt{Sys}}$, there is a $\sigma' \in \Gamma_{\texttt{Sys'}}$ such that $\sigma = \sigma'_{|CFset}$.

Lemma 5.2: Let $\sigma' \in \Gamma_{\texttt{Sys'}}$ and φ be a counting fluent expression. Then, for every position i: $\sigma', i \models (OBS \wedge fl_\varphi) \Rightarrow \sigma', i \models \varphi$.

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 [9]. An extension of LTSA that supports CFLTL and implements our model checking approach is available at [19].

VI. EVALUATION

In this section we evaluate our proposal answering these two main research questions:

RQ1: *is counting fluent temporal logic better suited than traditional fluent temporal logic for expressing properties in which the number of occurrences of certain events is relevant?*
RQ2: *how efficient is our model checking approach with respect to model checking equivalent fluent temporal logic specifications?*

In order to answer these questions, first we present several examples from the software engineering literature where the need of counting events arises and is addressed, in our opinion, unsatisfactorily. We show for each one, that counting fluents are well suited to formalise properties in which the occurrences of certain events must be measured, and allows for a cleaner separation between the behavioural models and the required properties, that otherwise would be tangled together to support verification.

We evaluate the quality of CFLTL specifications with respect to two metrics, *succinctness* [15] and *modifiability* [3]. To assess succinctness and modifiability we consider properties taken from the case studies and evaluate how concise CFLTL formulas are with respect to their original FLTL counterparts, and evaluating the complexity of introducing reasonable changes into the original specification, and the one developed by us in CFLTL.

Finally, we evaluate our model checking approach on several case studies that involve counting fluent specifications, under different configurations, and compare its efficiency against model checking equivalent fluent temporal logic specifications.

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:

$No\_Ignore\_Twice = \Box((\texttt{call} \wedge \Diamond \texttt{open}) \rightarrow$
$((\neg\texttt{atfloor} \wedge \neg\texttt{open}) \, \mathcal{U}(\texttt{open} \vee ((\texttt{atfloor} \wedge \neg\texttt{open}) \, \mathcal{U}$
$(\texttt{open} \vee ((\neg\texttt{atfloor} \wedge \neg\texttt{open}) \, \mathcal{U}(\texttt{open} \vee ((\texttt{atfloor} \wedge \neg\texttt{open}) \, \mathcal{U}$
$(\texttt{open} \vee (\neg\texttt{atfloor} \, \mathcal{U} \, \texttt{open}))))))))))$

Dwyer et al. argued that it is really difficult to convince oneself that this property captures exactly what is wanted. 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 CFLTL formula:

$ATFLOOR \equiv \langle\{\texttt{atfloor}\}, \{\}, \{\texttt{call}\}\rangle \text{ initially } 0$
$No\_Ignore\_Twice = \Box((\texttt{call} \wedge <>\texttt{open}) \rightarrow (ATFLOOR \leq 2 \, \mathcal{U} \, \texttt{open}))$

Timed Light: Let us consider another simple example, a *Timed Light* system presented in [24], 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 the counting fluent $T \equiv \langle\texttt{tick}, \{\}, \texttt{on}\rangle$, we are able to capture some interesting timed properties for this system:

$EventuallyOffOrPush = \Box(\texttt{on} \rightarrow \Diamond(T \leq 3 \wedge (\texttt{off} \vee \texttt{push})))$

Intuitively, counting fluent T counts the occurrences of tick, after the light is turned on. 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. Using the translation rules from bounded

FLTL into FLTL presented in [24], the above timed property can be specified in FLTL as follows:

$EventuallyOffOrPush = \Box(\texttt{on} \rightarrow ((\neg\texttt{tick} \vee \bigcirc(\neg\texttt{tick} \vee \bigcirc(\neg\texttt{tick} \vee$
$\bigcirc(\neg\texttt{tick} \vee \bigcirc(\neg\texttt{tick} \, \mathcal{W} \, (\texttt{off} \vee \texttt{push}))) \, \mathcal{W} \, (\texttt{off} \vee \texttt{push}))$
$\mathcal{W} \, (\texttt{off} \vee \texttt{push})) \, \mathcal{W} \, (\texttt{off} \vee \texttt{push})) \, \mathcal{W} \, (\texttt{off} \vee \texttt{push})))$

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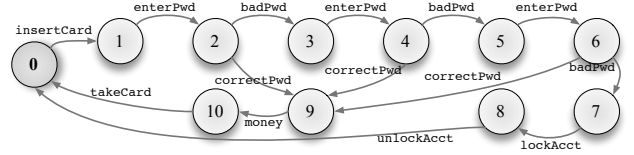

Fig. 3. LTS that models the ATM's behaviour.

Initially, the ATM requests the user to insert a card and enter the password (Pwd). The validity of the password is verified, and if this 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.

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 CFLTL property:

$ERRORS \equiv \langle\{\texttt{badPwd}\}, \{\}, \{\texttt{correctPwd}, \texttt{unlockAcct}\}\rangle \text{ initially } 0$
$NoWrongExtraction = \Box\neg(ERRORS \geq 3 \wedge \texttt{money})$

Intuitively, property *NoWrongExtraction* expresses that it cannot be the case that, after three consecutive wrong password insertions, the user extracts money. This property can be specified in FLTL in the following way:

$BadPass \equiv \langle\{\texttt{badPwd}\}, \{\texttt{correctPwd}, \texttt{unlockAcct}\}\rangle$
$NoWrongExtraction = \Box\neg(\texttt{badPwd} \wedge \bigcirc(BadPass \, \mathcal{U} \, (\texttt{badPwd} \wedge$
$\bigcirc(BadPass \, \mathcal{U}(\texttt{badPwd} \wedge \bigcirc(BadPass \, \mathcal{U} \, \texttt{money})))))$

Notice that, despite the fact that FLTL is expressive enough to specify this property, 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.

TCP Sliding Window: Consider the *TCP network protocol* [33], a protocol that provides reliable in-order delivery of packets in packet based data transmission. Figure 4 shows a LTS that models the behaviour of a single packet along a TCP network communication with its usual semantics. We represent the traffic in a network, combining the behaviour of various packets (PACKs). The TCP protocol has been improved many times, to optimize network transfer. In particular, the *TCP Sliding Window* protocol dynamically modifies the window size depending on the channel's reliability. The term *window* refers to the number of packets that can be sent without receiving their corresponding acknowledgements. This protocol considers that the window size is initially 1, and each time an ack is received, the window size is incremented by one

(i.e., the channel's reliability is increased), until MAX, the maximum size for the window, is reached. 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.
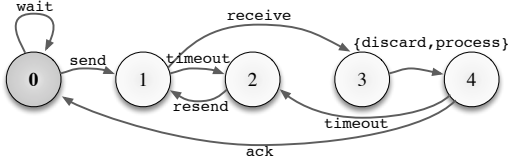

Fig. 4. Specification of TCP package behaviour.

An interesting property regarding this protocol is that the sender should always wait at most for MAX acks. Using CFLTL we can specify this property as follows:

$$ACK \equiv \langle\{\texttt{PACKs.send}\}, \{\texttt{PACKs.ack}\}, \{\}\rangle \text{ initially } 0$$
$$ACK\_less\_MAX = \Box(ACK \leq MAX)$$

A first attempt to capture this property in FLTL may be, after fixing the value of MAX (assume, for instance, $MAX = 5$) the nesting of temporal operators capturing the occurrence of $MAX + 1$ (six) successive send events without receiving an ack, similar to what was done for the elevator example.

However, this is weaker than desired, since it only captures a particular case of ACK_less_MAX. It does not consider, for instance, the case when 4 successive send occur, then an ack takes place, and finally 3 more send occur.

Our proposed FLTL solution to capture this property involves introducing N fluents (one per packet) to indicate if the packet has or has not been acknowledged. Then, the FLTL formula should encode the possibilities in which more than MAX sent packets have not been acknowledged.

$$ACK[p:1..N] \equiv \langle\{p.\texttt{send}\}, \{p.\texttt{ack}\}\rangle$$
$$ACK\_less\_MAX = \Box\neg(ACK[1..6] \vee ACK[2..7] \vee \ldots \vee ACK[(N-6)..N]$$
$$\vee(ACK[1] \wedge ACK[3..7]) \vee (ACK[1] \wedge ACK[4..8])\ldots)$$

Notice that we have $\binom{N}{MAX+1}$ cases in which property $ACK \leq MAX$ is violated. Thus, for instance, for 10 packets and $MAX = 5$, the FLTL formula will contain 210 disjunctions. Dealing with this "counting property" manually in FLTL is clearly impractical, and definitely error prone.

The above property is actually a simplification of a stronger intended property of the protocol, namely that it is always the case that the number of packets not acknowledged is bounded by the *current* window size.

It is important to observe that this stronger property needs to refer to a *dynamic* value, a value that changes as the system is executing. So, let us define $WINDOW$, 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:

$$WINDOW \equiv \langle\{\texttt{PACKs.ack}\}, \{\texttt{PACKs.timeout}\}, \{\}\rangle \text{ initially } 1$$
$$ACK\_less\_WINDOW = \Box(ACK \leq WINDOW)$$

In contrast to the previous examples, we cannot express this property in FLTL nesting temporal and logical operators.

Producers and Consumers: This is a classic concurrency problem [26], 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 *full* models the number of values in the buffer (initially 0); and a semaphore *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).

In order to capture the correct synchronisation between the elements of the system, we can specify the following property:

$$CountP \equiv \langle\{\texttt{empty.V}\}, \{\texttt{empty.P}\}, \{\}\rangle \text{ initially } C$$
$$CountC \equiv \langle\{\texttt{full.V}\}, \{\texttt{full.P}\}, \{\}\rangle \text{ initially } 0$$
$$CorrectSynchronisation = \Box(CountP + CountC \leq C)$$

CorrectSynchronisation expresses that the number of occupied spaces plus the number of free spaces (*CountP* + *CountC*) in the buffer should always be less than the capacity C. The sum 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 synchronising, 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.

Further examples: In addition to the presented examples, we report here 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. In [5]: count the number of times the Store reports that a requested item is unavailable; [25]: In an auctioneer web service, "a user can only bid at most 3 times within each of his logins"; [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]: In a 5 minutes interval the customer can only do 3 failed payment trials.

A. Succinctness and Modifiability Evaluation

Succinctness [15] is a measure used for comparing how concisely two logics can specify certain properties. Usually, it is applied to logics with the same expressive power. For those properties that can be expressed both in FLTL and CFLTL mentioned earlier, we evaluate how succinctly these are expressed in each of the logics. Regarding the modifiability comparison between FLTL and CFLTL, we propose reasonable changes to the original requirements for the above introduced case studies and evaluate the complexity of the revised formulas in terms of succinctness. Examples of reasonable changes to be introduced are the need to increase the number of times the elevator may visit the floor without opening its doors in

the elevator model, or changing the capacity of the bridge and the amount of cars in the SLB model.

Table I summarises the comparison between the CFLTL and FLTL formulas capturing the above mentioned properties of our case studies. For the evaluation of succinctness, the considered attributes are: the number of counting/propositional fluents and events involved in the formulas; the maximum nesting level of temporal operators; and the number of temporal and logical operators used in the formulas.

TABLE I
COMPARING FLTL & CFLTL ON SUCCINCTNESS/MODIFIABILITY.

Model/prop. changes	Fluent LTL				Counting Fluent LTL			
	Fl./e	Nes.	T.Op.	L.Op.	Fl./e	Nes.	T.Op.	L.Op.
model: **elevator**, property: *No_Ignore_C_times*, changes = **C**								
2	3	5	7	17	3	1	3	2
6	3	13	15	45	3	1	3	2
7	3	15	17	52	3	1	3	2
model: **timed light**, property: *EventuallyOffOrPush*								
-	4	6	7	14	4	1	2	3
model: **SLB**, property: *Capacity_Safe*, changes = **Cars(each color)/Capacity**								
6/3	12	0	1	116	1	0	1	0
7/4	14	0	1	205	1	0	1	0
8/4	16	0	1	555	1	0	1	0
model: **ATM**, property: *WrongExtraction*								
-	3	6	7	4	2	0	1	1
model: **TCP**, property: *ACK_less_Max*, changes = **Pack/Window size**								
10/6	10	0	1	840	1	0	1	0
10/7	10	0	1	360	1	0	1	0
11/5	11	0	1	2772	1	0	1	0
11/6	11	0	1	2310	1	0	1	0
model: **TCP**, property: *ACK_less_WINDOW*								
-	inexpressible				2	0	1	0
model: **Prod-Cons**, property: *CorrectSynchronisation*								
-	inexpressible				2	0	1	0

It can be observed from the table that the difference in the number of fluents and logical operators required in some formulas, when comparing FLTL with CFLTL, is significant. For instance, when specifying FLTL properties like $CAPACITY\_SAFE$ and ACK_less_MAX, events involving different items need to be appropriately distinguished (via fluents), and their combinations handled as a kind of bit array, to capture the counting of events. Also, when the occurrences of events need to be counted in FLTL (e.g., three successive wrong password insertions in the ATM model), temporal operators have to be nested to capture the counting. Complex nestings make formulas harder to read and thus more difficult to contrast with the properties being captured. CFLTL formulas, on the other hand, require fewer fluent definitions, and avoid nesting temporal and logical operators by counting through counting fluents.

Table I also shows that property changes cannot be easily incorporated in FLTL (requiring further nestings and making formulas more complex), as opposed to the case of CFLTL, where these required just minor formula modifications (some changes were incorporated without altering the CFLTL formula, only changing the constant values counting fluents were compared with, e.g., the maximum window size). This is essentially due to the fact that CFLTL allowed us to express properties involving counting events without the need to manually instrument the models, making model/formula modifiability easier to achieve.

Finally, as mentioned earlier, counting fluents enabled us to capture a system property involving a dynamic value (a numeric value that changes during system execution), that cannot be expressed in FLTL.

B. Model Checking Evaluation

To evaluate the performance of our model checking approach introduced in Section V, we use LTSA model checking on some of the models and properties presented above. We compare each original FSP model with the property expressed in FLTL (when expressible), with the corresponding automatically generated FSP model, that reduces the property expressed in CFLTL to FLTL model checking. To evaluate the scalability of our model checking technique in comparison with FLTL model checking, we assess both model checking approaches on our case studies as values in models and formulas (in particular, values relevant to counting) are increased (these value changes are similar to those used in the modifiability evaluation). Table II presents the results in terms of state space required for formula and system representations, and the maximum memory and total time required for verification.

The translation and verification were performed using an Intel Core 2 Duo 2Ghz processor, with 3GB DDR2 memory, running GNU/Linux. As it can be observed in Table II, our approach is in general more efficient and scales better than FLTL model checking on the original models. Notice that the total time reported in the CFLTL analyses includes the *fluent overflow* checking, which is roughly the same time spent in the CFLTL formula verification (i.e., the actual CFLTL formula verification time is approximately half the total time reported).

TABLE II
MODEL CHECKING EVALUATION.

Model/ prop. changes	Fluent LTL				Counting Fluent LTL			
	Space 2^x		Mem.	Time	Space 2^x		Mem.	Time
	For.	Sys.	MByte	sec.	For.	Sys.	MByte	sec.
model: **elevator**, property: *No_Ignore_C_times*, changes = **C**								
5	8	10	7.3	77.53	9	16	1.4	0.71
6	9	10	9.4	868.17	9	16	2.0	0.95
7	not enough memory				9	16	6.1	1.57
5000	not enough memory				9	25	312.5	438.60
8000	not enough memory				9	26	709.9	2209.49
model: **SLB**, property: *Capacity_Safe*, changes = **Cars(each color)/Capacity**								
6/3	14	42	232.3	171.15	5	44	3.3	1.63
7/4	16	48	733.8	4972.94	5	47	3.5	2.00
8/4	not enough memory				5	51	4.1	2.26
100/50	not enough memory				5	255	806.6	1171.35
150/100	not enough memory				5	371	2423.0	12966.32
model: **TCP**, property: *ACK_less_Max*, changes = **Pack/Window size**								
10/6	12	46	976.1	262.91	5	46	1148.1	1107.97
10/7	12	46	1761.8	1938.25	not enough memory			
10/8	not enough memory				not enough memory			
11/5	13	48	864.4	2182.54	5	48	641.2	395.29
11/6	13	50	1369.2	1690.64	5	49	1967.4	5669.30
model: **TCP**, property: *ACK_less_WINDOW*, changes = **Pack/Window size**								
15/12	inexpressible				5	66	7.4	79.06
20/18	inexpressible				5	83	17.0	723.75
25/23	inexpressible				5	99	29.3	3513.01
model: **Prod-Cons**, property: *CorrectSynchronisation*, changes = **Prod/Cons/Buffer**								
7/5/15	inexpressible				5	46	1137.8	1046.49
7/5/20	inexpressible				5	49	1612.1	2947.40
7/5/30	inexpressible				5	50	2214.6	15495.29

It is interesting to observe that FLTL properties make the verification state space grow exponentially, as the size of models/formulas is increased (with respect to values relevant

to counting events). On the other hand, CFLTL properties maintain the state space more stable as the size of models/formulas is increased, but with the extra cost of extending the models with monitors as explained in Section V. This explains the improved efficiency and scalability of our CFLTL model checking approach compared to FLTL verification. We noticed that, in the latter, the Büchi automata generation is very time consuming, affecting the efficiency of the approach (an issue that is more prominent when formulas have nested until operators, as in the elevator and SLB models).

The absence of *fluent overflows* in the experiments is due to the fact that we manually selected appropriate tight values for counting fluents' bounds and scopes (suggested by inherent limits present in the models), to prevent these overflows.

An additional benefit that we observed in the experiments, related to the use of counting fluents, is an easier counterexample interpretation; when a property is violated in our CFLTL verification approach, the counterexample explicitly reports counting fluent modifications as indexed events, helping in understanding the implications of the change (and the value of the counting fluent) in the formula that involves it.

More detailed information regarding the model checking evaluation, including the sources of all original and generated models and the output of the LTSA runs for the experiments, can be found in [9].

VII. RELATED WORK

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], [4]. Among them, Counting CTL [22] and Counting LTL [21] 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) [32]. 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 range of techniques and tools. However, such an advantage also imposes limitations in expressiveness. For instance, it is impossible to compare among occurrences of events, and hence a simple example such as the Single Lane Bridge cannot be described as properties relating occurrences of blue and red cars.

In the AI planning [13], [23] community a technique to capture numeric values with fluents has been proposed, namely additive fluents [23]. 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 I, the approach introduced in this article is closely related to fluent temporal logic (FLTL) [14]. FLTL enables specifying LTL properties of event-based systems, where propositions are capture via *fluents*. As opposed to the boolean nature of fluents, our counting fluents are numerical values, that enable us to enumerate event occurrences, and as previously described, leads to greater expressive power, compared to FLTL. Another related extension to FLTL is that presented in [24], in which special temporal operators for modelling timed properties of discrete-time event-based models are introduced (basically, temporal operators are equipped with bounds for counting the progress of time). The work in [24] shows the necessity for counting the occurrences of certain events, although the proposal is limited to counting a particular event, namely `tick`, that models time progress.

VIII. CONCLUSIONS

We have introduced Counting Fluent Linear Temporal Logic, an extension of FLTL with the concept of *counting fluent*. Counting fluents represent numerical variables that generalise the boolean character of fluents, and provide the flexibility to specify naturally many properties of reactive systems in which the number of occurrences of certain events is relevant. We proved that the logic is strictly more expressive than LTL (and FLTL), and also that is undecidable. We also equipped CFLTL with a sound but incomplete (due to the undecidability of the logic) model checking approach, that allows us to analyse CFLTL formulas over finite state models under user defined scopes. Moreover, we performed a thorough evaluation of multiple aspects of our proposal. We argued 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 an evaluation on CFLTL considering two metrics, succinctness and modifiability, to compare the same properties specified in CFLTL and FLTL, and showing that CFLTL leads to simpler, more understandable and more modifiable formulas. We also evaluated our model checking approach by comparing the efficiency of our CFLTL verification with standard FLTL verification on equivalent properties. The results show that our approach leads to an improved efficiency, and better scalability.

A tool that extends LTSA to support CFLTL and our model checking approach can be obtained from [19]. The tool is a prototype, and we are currently working on improving it. In addition, we are working on the formal underpinnings of the introduced logic, studying the details of the undecidability of CFLTL. We also plan to study which fragments of CFLTL are decidable, and subject to sound and complete model checking, as an alternative to our incomplete model checking approach, and to further explore the relationship between our logic and Counting LTL.

REFERENCES

[1] L. Aceto, A. Ingólfsdóttir, K. Larsen and J. Srba, *Reactive Systems: Modelling, Specification and Verification*, Cambridge University Press, 2007.

[2] R. Alur and T. Henzinger, *A really temporal logic*, J. ACM, Vol. 41, No. 1, pp. 181-203, 1994.

[3] F. Bachmann, L. Bass, and R. Nord, *Modifiability tactics*, Technical report CMU/SEI-2007-TR-002. Software Eng. Inst., 2007.

[4] C. Baier and J.P. Katoen, *Principles of Model Checking*, MIT Press, 2008.

[5] F. Barbon, P. Traverso, M. Pistore and M. Trainotti, *Run-Time Monitoring of Instances and Classes of Web Service Compositions*, in Proc. of ICWS'06, IEEE, pp. 63-71, 2006.

[6] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea and P. Spoletini, *Validation of web service compositions*, in Software, IET, pp. 219-232, 2007.

[7] D. Bianculli, C. Ghezzi and P. San Pietro, *The Tale of SOLOIST: A Specification Language for Service Compositions Interactions*, in Proc. of FACS '12, pp. 55-72, 2012.

[8] E. Clarke, O. Grumberg and D. Peled, *Model Checking*, MIT Press, 2000.

[9] Counting Fluent Linear Time Temporal Logic Experimental Results ICSE 2015
http://dc.exa.unrc.edu.ar/staff/gregis/ICSE2015

[10] R. Darimont and A. van Lamsweerde, *Formal Refinement Patterns for Goal-Driven Requirements Elaboration*, in Proc. of FSE'96, ACM, pp. 179-190, 1996.

[11] V. Diekert and P. Gastin, *First-order definable languages*, Logic and Automata, pp. 261-306, 2008.

[12] M. Dwyer, G. Avrunin and J. Corbett, *Patterns in Property Specifications for Finite-state Verification*, in Proc. of ICSE'99, ACM, pp. 411-420, 1999.

[13] E.Erdem and A. Gabaldon, *Representing Action Domains with Numeric-Valued Fluents*, Logics in Artificial Intelligence Vol. 4160, pp 151-163, 2006.

[14] D. Giannakopoulou and J. Magee, *Fluent Model Checking for Event-based Systems*, in Proc. of ESEC/FSE'03, ACM, pp. 257-266, 2003.

[15] M. Grohe and N. Schweikardt, *The succinctness of first-order logic on linear orders*, Logical Methods in Computer Science, Vol. 41, No. 1, 2005.

[16] C. A. R. Hoare, *Communicating sequential processes*, Prentice-Hall 1985.

[17] S. Kallel, A. Charfi, T. Dinkelaker, M. Mezini and M. Jmaiel, *Specifying and Monitoring Temporal Properties in Web Services Compositions*, in Proc. of ECOWS'09, IEEE, pp. 148-157, 2009.

[18] H.W. Kamp, *Tense Logic and the Theory of Linear Order*. PhD thesis, University of California, USA (1968).

[19] Labelled Transition System Analyser with Counting Fluents Support
http://sourceforge.net/projects/cf-ltsa/

[20] A. van Lamsweerde, A. Dardeene, D. Delcourt and F. Dubisy, *The KAOS Project: Knowledge Acquisition in Automated Specification of Software*, in Proc. of AAAI Spring Symposium Series, Track: "Design of Composite Systems", Stanford University, pp. 59-62, 1991.

[21] F. Laroussinie, A. Meyer and E. Petonnet, *Counting LTL*, in Proc. TIME '10, IEEE, pp. 51-58, 2010.

[22] F. Laroussinie, A. Meyer and E. Petonnet, *Counting CTL*, in Proc. FOSSACS '10, LNCS, vol. 6014, pp. 206-220, 2010.

[23] J. Lee and V. Lifschitz, *Describing additive fluents in action language $\mathcal{C}+$*, in Proc. IJCAI '03, pp. 1079-1084, 2003.

[24] E. Letier, J. Kramer, J. Magee and S. Uchitel,*Fluent Temporal Logic for Discrete-Time Event-Based Models*, in Proc. of ESEC/FSE'05, ACM, pp. 70-79, 2005.

[25] Z. Li, J. Han and Y. Jin, *Pattern-Based Specification and Validation of Web Services Interaction Properties*, in Proc. of ICSOC'05, LNCS, pp. 73-86, 2005.

[26] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, John Wiley & Sons, 1999.

[27] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems - Specification -*, Springer, 1991.

[28] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems - Safety-*, Springer, 1995.

[29] N. Medvidovic and R. Taylor, *A classification and comparison framework for software architecture description languages*, IEEE Transactions on Software Engineering 26(1), IEEE, 2000.

[30] R. Miller and M. Shanahan, *The Event Calculus in Classical Logic - Alternative Axiomatisations*, Linkoping Electronic Articles in Computer and Information Science, Vol. 4, No. 16, pp. 1-27, 1999.

[31] R. Milner, *Communication and Concurrency*, Prentice-Hall, 1989.

[32] M. Pradella, A. Morzenti and P. San Pietro, *The Symmetry of the Past and of the Future: Bi-infinite Time in the Verification of Temporal Properties*, in Proc. of ESEC-FSE '07, pp. 312-320, 2007.

[33] A. Tanenbaum and D. Wetherall, *Computer Networks*, Prentice-Hall, 2010.

[34] S. Uchitel, J. Kramer and J. Magee, *Synthesis of Behavioral Models from Scenarios*, in Proc. of IEEE Transactions on Software Engineering, Vol. 29, pp. 99-115, 2003.