# Counting Fluent Temporal Logic

Technical Report

In this technical report we aim to describe the details of the proposed model checking algorithm for Counting Fluent Temporal Logic.

## Motivating Example

We will start to motivate our work in this paper with the Single Lane Bridge Problem (SLB), a modelling problem introduced in [6] (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; more precisely, red cars will move in one direction, while blue cars will move in the opposite one. The following FSP model is a specification of this problem.

```
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 is the red count, nb the blue count
      (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 [6], we need to express whether there is at least one car of each colour on the bridge. Following the solution presented in [6, 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:

$$\begin{aligned} ONEWAY = \Box\neg((&RED[1] \vee RED[2] \vee \ldots \vee RED[N])\wedge \\ &(BLUE[1] \vee BLUE[2] \vee \ldots \vee BLUE[N])) \end{aligned}$$

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 [6]). 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 is the red count, nb the blue count
  (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*:

$$\begin{aligned}CAPACITY\_SAFE = \Box\neg(&(RED[1] \wedge RED[2] \wedge RED[3])\vee\\ &(RED[1] \wedge RED[2] \wedge RED[4])\vee\\ &(RED[1] \wedge RED[3] \wedge RED[4])\vee\\ &(RED[2] \wedge RED[3] \wedge RED[4])\vee\\ &(BLUE[1] \wedge BLUE[2] \wedge BLUE[3])\vee\\ &(BLUE[1] \wedge BLUE[2] \wedge BLUE[4])\vee\\ &(BLUE[1] \wedge BLUE[3] \wedge BLUE[4])\vee\\ &(BLUE[2] \wedge BLUE[3] \wedge BLUE[4]))\end{aligned}$$

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:

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

Now let us 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:

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

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 our validation Section. 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.

## 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 [3], 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* $N$.

*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 $\Psi$ of counting fluents and $cFl_1, cFl_2 \in \Psi$, a valid counting expression $\phi$ is defined as follows:

$$\begin{aligned} \phi ::= cFl_1 \sim c \mid cFl_1 \sim cFl_2 \mid cFl_1 \sim cFl_2 \pm c \\ \text{s.t., } c \in \mathbb{N} \text{ and } \sim \in \{=, >, <\}. \end{aligned}$$

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 $\phi$ 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 $\neg\varphi_1$, $\varphi_1 \vee \psi_2$, $\varphi_1 \wedge \psi_2$, $\bigcirc\varphi_1$, $\varphi_1 \mathcal{U} \varphi_2$, and the usual derived definitions for $\Box\varphi_1$ and $\Diamond\varphi_1$.

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 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 [3]. Then, to obtain a complete interpretation of CFLTL formulas, we define the semantics for the counting expressions as follows:

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

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 $cFl_1 \sim cFl_2 \pm 0$.

### 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 [2]. 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 [4] 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.

## A Model Checking approach for CFLTL

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 [1] 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 [3]. 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 unboundness 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 [6]. 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):

⟨*CFluentDef*⟩ ::=
 **'cfluent'** ⟨*fluent_name*⟩ ⟨*fluent_bounds*⟩ **'='**
  **'<'**⟨*incremental_events_set*⟩ **','** ⟨*decremental_events_set*⟩ **','**
   ⟨*reset_events_set* ⟩ **'>'** **'initially'** ⟨*initial_value* ⟩
⟨*fluent_bounds*⟩ ::=
 ( '[' | '(' ) ⟨*min_value*⟩ **'..'** ⟨*max_value*⟩ ( ']' | ')' )

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

## 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 the *TCP network protocol*. 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:

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

Notice that $ACK$ may be not bounded in a model an a violation to $ACK\_less\_MAX$ could be found. 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.

## Model Checking

Let `Sys` and $\phi$ be a FSP specification of a system and a CFLTL property, respectively, and suppose that $\phi$ contains fluent expressions $\epsilon_1, \ldots, \epsilon_n$ that refer to counting fluents $cFl_1, cFl_2, \ldots, cFl_m$. In order to perform the verification process using LTSA, our approach generates a new FSP process `Sys'` and a FLTL formula $\phi'$, such that `Sys'` incorporates the monitor process that updates the values of the counting fluents and $\phi'$ encodes the propositional fluents associated to each counting expression. The construction of `Sys'` and $\phi'$ ensures that every counterexample for $\phi'$ in `Sys'` is a counterexample for $\phi$ in `Sys`. Formally, $\texttt{Sys}' \not\models_{FLTL} \phi' \Rightarrow \texttt{Sys} \not\models_{CFLTL} \phi$. Below, we describe our approach, consisting of constructing the monitor and the encoded formula $\phi'$.

### Monitors for Counting Fluents

`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, `CfEvs` 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.

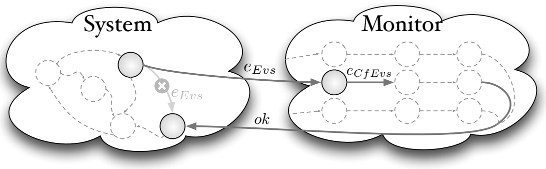

Figure 1: Behavioural view of `Sys'`.

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

Figure 2: FSP spec. for `SYNCH`.

Intuitively, a monitor keeps track of the values of the counting fluents (within its bounds/scopes) that appear in a counting expression. In the case of a unary expression $cFl \sim c$, the monitor records the value of counting fluent $cFl$, and for binary expressions ($cFl_1 \sim cFl_2 \pm c$), the monitor records the values of counting fluents $cFl_1$ and $cFL_2$.

To achieve this, the monitor process records the counting fluent values by means of parameters, one per counting fluent $cFl_1, \ldots, cFl_m$. Then, each parameter is initialised with the initial value of the corresponding counting fluent: CFmon = CFmon$_B[I_1]\ldots[I_m]$. Process CFmon monitors the occurrence of system events involved in the definition of the counting fluents, and triggers particular events that update the values of the counting expressions.

First, consider the case in which the parameters values are not in a boundary situation, i.e. they are strictly within their limits (lower and upper bounds/scopes). Given values $v_1, \ldots, v_m$ that represent a valid state of the CFmon process, we generate a process case for each monitored event $e$, as follows:

```
| when ($\gamma$) $e$ ->$\epsilon_1$[$v'_{\epsilon_1}$] ->$\epsilon_2$[$v'_{\epsilon_2}$] ->... ->$\epsilon_n$[$v'_{\epsilon_n}$] ->ok ->CFmon_B [$v'_1$] ... [$v'_m$]
```

The guard $\gamma$ ensures the values are not in a boundary situation. When the monitored event $e$ occurs, then the monitor process triggers $n$ events ($\epsilon_i[v_{\epsilon_i}]$) to update the values of each counting expression $\epsilon_1, \ldots, \epsilon_n$.

In case of an unary expression (e.g., $\epsilon = cFl \sim c$), the triggered event has the form $\epsilon[v'_{cFl}]$, where $v'_{cFl}$ is the value of the counting fluent associated to $\epsilon$. For a binary expression (e.g., $\epsilon = cFl_i \sim cFl_j \pm c$), the value of the triggered event represents the difference between the counting fluents values related in the expression, i.e., $\epsilon[v'_{cFl_i} - v'_{cFl_j}]$. As detailed below, these events are used in the definition of the propositional fluents that hold the truth values for each counting expression.

The new values for $v'_1 \ldots v'_m$, the parameters of the monitor, are calculated in terms of the membership of the event $e$ to the incremental, decremental or reset event sets of the corresponding counting fluent definitions.

Consider now the case where some parameter is in a boundary situation, i.e., one of $v_1, \ldots, v_m$ reaches its lower (upper) limit. Then, for each expression $\epsilon_i$ and decremental (incremental) event $e$ of it, we generate a process case for 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.

In order to illustrate a monitor process generated by our model checking approach, let us consider the specification of the SLB problem presented previously, 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 the following:

```
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]).
```

### Encoding CFLTL formulas

In order to verify an CFLTL formula $\phi$ using LTSA, we encode it as an FLTL formula $\phi'$ which captures the truth values of the counting fluent expressions with propositional fluents. Thus, we define fluents $fl_1, \ldots, fl_n$, one per counting expression, such that, if $\epsilon_i$ holds at a position, then so does $fl_i$.

The form of the counting expression $\epsilon_i$ (i.e., unary or binary) determines the sets of events that define the propositional fluent $fl_i$.

- If $\epsilon_i = cFl \sim c$, then the propositional fluent that captures the truth value of the expression is the following: $fl_i \equiv \langle \epsilon_i[\sim c], \epsilon_i[\nsim c] \rangle$, where $\epsilon_i[\sim c]$ is the set of events of $\epsilon_i$ satisfying the relation $\sim c$ under bounds $l_{\epsilon_i}$ and $u_{\epsilon_i}$. For instance, for the relation $\leq 2$ under 0 and 3 as lower and upper bounds, respectively, $\epsilon_i[\leq 2]$ is the set $\{\epsilon_i[0], \epsilon_i[1], \epsilon_i[2]\}$.
- In case of binary expressions, i.e., $\epsilon_i = cFl_1 \sim cFl_2 \pm c = cFl_1 - cFl_2 \sim \pm c$, the generated fluent has the form: $fl_i \equiv \langle \epsilon_i[\sim 0 \pm c], \epsilon_i[\nsim 0 \pm c] \rangle$. Note that, in these cases, the equality of two fluents is captured by the event $\epsilon_i[0]$, so the relations can be obtained by analysing the events that express the monitored difference. As an example, if $\epsilon_i$ is the expression $cFl_1 \leq cFl_2 + 1$ under bounds $-2$ and 2, the set of events that enable the fluent $fl_i$ is $\{\epsilon_i[-2], \epsilon_i[-1], \epsilon_i[0], \epsilon_i[1]\}$.
- Finally, the initial value for $fl_i$ is *true* if the counting expression $\epsilon_i$ holds with the initial counting fluent values, and *false* otherwise.

Notice that there exist some states in `Sys'` in which $\phi'$ must not be evaluated, i.e., 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 been overflowed. With this notion, the last step of the construction of $\phi'$ is based on the translation introduced in [5, Subsection 5.3] to guarantee the exclusion of the non-observable states in the analysis of the validity of $\phi$ in a model. For instance, if $\phi =$ `[]`$\varphi$, then $\phi' =$ `[]`$(OBS \rightarrow \varphi)$.

Following with the SLB example, we can now encode the CFLTL formula *SAFE_CAPACITY* (where C instantiated with value 2) as follows:

```
fluent CARS_ON_BRIDGE_L_3 = <carsOnBridge[0..2],
                             carsOnBridge[3..4]> initially True
fluent OK =<ok,MonEvs> initially True
assert SAFE_CAPACITY=[]((OK &&!F_overflow)->CARS_ON_BRIDGE_L_3)
```

### Verification

Suppose that the encoded formula $\phi'$ 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 `REACH_OVERFLOW = [](!F_overflow)`. If

`REACH_OVERFLOW` 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 $\phi'$ exists. That is, our approach guarantees in this case the validity of property $\phi$ 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.

## Lemmas

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 $\sigma$ over $Evs$, $\sigma \;|_A$ returns the trace obtained by ignoring the occurrences of events $e \in A$ in $\sigma$. 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 $CFset = CfEvs \cup \{\texttt{ok}\} \cup \{\texttt{fluent\_overflow}\}$ contains all events performed by the monitor `CFmon`. Then, the following lemmas hold.

**Lemma 1** *For every $\sigma \in \Gamma_{\mathtt{Sys}}$, there is a $\sigma' \in \Gamma_{\mathtt{Sys'}}$ such that $\sigma = \sigma'_{|_{CFset}}$.*

**Proof** Let $\sigma = e_1 \to \ldots \to e_n$ be a trace in $\Gamma_{\texttt{Sys}}$, in order to show that exist some $\sigma' \in \Gamma_{\texttt{Sys'}}$ such that $\sigma = \sigma'_{|_{CFset}}$, consider the following two cases:

- if there is no monitored event executed in $\sigma$, i.e. every $e_i \notin MonEvs$, then the existence of $\sigma' = \sigma$ is inmediate due to the no interference (synchronisation) of `CFmon`, the `SYNCH` definition and the parallel composition properties.
- Now, for every $e_i \to e_{i+1}$ sub-trace of $\sigma$, where $e_i \in MonEvs$, there is a sub-trace

  $$e_i \to \epsilon_1[v'_{\epsilon_1}] \to \ldots \to \epsilon_k[v'_{\epsilon_k}][\to \texttt{fluent\_overflow}] \to \texttt{ok} \to e_{i+1}$$

  of some $\sigma' \in \Gamma_{\texttt{Sys'}}$ due to the following reasons: the disjunction of the conditions in `CFmon` for every monitored event of the system is *true*. Moreover, the monitor behaviour serialises, without interruption (`SYNCH`), the updating events $\epsilon_i[v'_{\epsilon_i}] \in CfEvs$ for the $k$ fluent expressions values. Finally, the monitor ends with an `ok` event (including the case when `fluent_overflow` is triggered), and continues with $e_{i+1}$ as the `SYNCH` process guarantee.

  By applying the $|_A$ function to these sub-traces, we obtain that

  $$\sigma = \sigma'_{|_{CFset}}$$

**Lemma 2** *Let $\sigma' \in \Gamma_{\mathtt{Sys'}}$ and $\varphi$ a counting fluent expression, for all position $i$:*

$$\sigma', i \models (OBS \wedge fl_\varphi) \Rightarrow \sigma', i \models \varphi$$

Lemma 2 expresses that, under bounded situations, the truth value of a counting fluent expression is captured by the corresponding propositional fluent generated over the monitored system.

Recall that $OBS \equiv \texttt{OK} \wedge \neg\texttt{F\_overflow}$, where OK indicates if the monitor finished the updating process of the counting fluent values, and F_overflow alerts if a overflowed state has been reached. For states which are not observables, i.e. $OBS$ is false, the implication trivially holds.

Consider now some state at the position $i$ that satisfies the $OBS$ condition. For $i = 0$ the formula trivially holds by the initialization definition of $fl_\varphi$, i.e. it is initialized with the evaluation of the expression instantiated with the initial counting fluent values present in $\varphi$. Whether $i > 0$, to simplify the proof, let us consider first the case of unary expressions, i.e. expressions such as $cFl \sim c$. If $OBS \wedge fl_\varphi$ holds, then there is a sequence of updating events of the form $\varphi[v'_{cFl}]$, where the last value $v = v'_{cFl}$ belongs to the activating set of the propositional fluent $fl_\varphi$, i.e. $v$ fulfil the formula $v \sim c$.

By the monitor's construction, these updating events correspond to the a sequence of monitored events occurrences. Starting from the monitor process with the $N_{cFl}$ value, and ignoring prefix sub-traces ending with a reset event $e \in R_{cFl}$ (in which case the parameter value go backs to the value $N_{cFl}$), there is a final sub-trace which has the necessary monitored events $e_1, \ldots, e_n \in I_\varphi \cup D_\varphi$ before the triggering of $\varphi[v]$. As consequence of the construction of Cfmon and SYNCH, $N_{cFl} + \#e_i \in I_\varphi - \#e_j \in D_\varphi = v$ which is the definition of the semantic value of $cFl$ and as consequence $cFl \sim c$ as desired.

For a binary expression (e.g., $\varphi = cFl_i \sim cFl_j \pm c$), if $OBS \wedge fl_\varphi$ holds, then there is a sequence of updating events that triggers the event $\varphi[v]$, where $v = v_i - v_j$ is the difference between the corresponding values of the counting fluents associated with $\varphi$, and $v$ fulfil de formula $v \sim 0 \pm c$. By a similar reasoning as for unary expression, we have a set of system events occurrences $e_{i_1}, \ldots, e_{i_n} \in I_{cFl_i} \cup D_{cFl_i}$ and $e_{j_1}, \ldots, e_{j_n} \in I_{cFl_j} \cup D_{cFl_j}$ such that $v_i = N_{cFl_i} + \#e \in I_{cFl_i} - \#e \in D_{cFl_i}$ and $v_j = N_{cFl_j} + \#e \in I_{cFl_j} - \#e \in D_{cFl_j}$ which satisfies the meaning of $\varphi$.

## References

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

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

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

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

[5] 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.

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