# CLTSA: Labelled Transition System Analyser with Counting Fluent support

Germán Regis
University of Río Cuarto, Argentina

Renzo Degiovanni
University of Río Cuarto & CONICET, Argentina

Nicolas D'Ippolito
University of Buenos Aires & CONICET, Argentina

Nazareno Aguirre
University of Río Cuarto & CONICET, Argentina

## ABSTRACT

In this paper we present CLTSA (Counting Fluents Labelled Transition System Analyser), an extension of LTSA (Labelled Transition System Analyser) that incorporates counting fluents, a useful mechanism to capture properties related to counting events. Counting fluent temporal logic is a formalism for specifying properties of event-based systems, which complements the notion of *fluent* by the related concept of *counting fluent*. While fluents allow us to capture boolean properties of the behaviour of a reactive system, counting fluents are numerical values, that enumerate event occurrences.

The tool supports a superset of FSP (Finite State Processes), that allows one to define LTL properties involving counting fluents, which can be model checked on FSP processes. Detailed information can be found at http://dc.exa.unrc.edu.ar/tools/cltsa.

## CCS CONCEPTS

• **Theory of computation → Modal and temporal logics**; **Verification by model checking**; • **Software and its engineering → Specification languages**;

## KEYWORDS

Model checking, Temporal Logic, Specification and Verification

**ACM Reference format:**
Germán Regis, Renzo Degiovanni, Nicolas D'Ippolito, and Nazareno Aguirre. 2017. CLTSA: Labelled Transition System Analyser with Counting Fluent support. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 5 pages.
https://doi.org/10.1145/3106237.3122828

## 1 INTRODUCTION

The increasingly rich set of tools and techniques for software analysis offers unprecedented opportunities for helping software developers in finding program bugs, and discovering flaws in software models [4, 11, 12]. 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 software systems, most notably parallel and concurrent systems.

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 [5], CCS [10] and FSP [9], some mechanisms have been proposed to capture state properties in event-based systems, too. Through the notion of *event*, which is used as a means to represent components behaviour and interaction on event-based formalisms, *fluents* are proposed in [1] in order to enable the use of temporal logic for specifying properties of event-based systems. Fluents are propositional variables that allow one to capture state propositions in these systems, in terms of activating and deactivating events. Based on the fluent concept and with the aim of dealing with properties of reactive systems in which the number of occurrences of certain events is relevant, the notion of *counting fluent* was introduced in [13]. As opposed to the boolean nature of a fluent, a counting fluent represents a numerical value that enumerates event occurrences in terms of incrementing, decrementing and resetting events.

Of course, a convenient language for specifying system properties is not enough: such a language must be accompanied by powerful tool support. In [13], a prototypical tool was presented to support counting fluents. Given an FSP model of a reactive system, the tool allowed one to specify counting fluents that monitored the behaviour of the system and to use them as part of counting expressions for specifying counting properties. Moreover, the tool could also model check these properties. Intuitively, given the counting fluent temporal formula, and the limits for each counting fluent, the approach of [13] automatically generated a monitoring process to capture the valuation of the counting expressions, and then reduced the problem of model checking the counting property to model checking an "equivalent" propositional temporal property by using LTSA [9].

In this tool demonstration paper, we present CLTSA (Counting Fluents Labelled Transition System Analyser), a new tool that implements a direct model checking technique for counting fluents linear temporal logic, building upon a traditional LTS model verification. In contrast with the approach in [13], this new technique implements an automata representation for counting expressions, improving the efficiency and scalability of the analysis. In addition,

Partially supported by ANPCyT PICT-2015-0586, PICT-2015-2088, PICT-2012-1298.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany*
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3122828

as an advantage of the automata representation for counting expressions, CLTSA supports richer counting properties than those supported by [13]. For instance, CLTSA allows expressions with any number of counting fluents and a wide range of arithmetical operators, including addition, subtraction, multiplication, integer division and remainder.

**Contribution**. CLTSA supports: • Counting fluents definition in terms of system events. • Specification of LTL properties that involve counting fluents and counting expressions that may involve a wide range of arithmetical expressions. • Definition of different kinds of limits for counting fluents, required by the model checking approach. • An automated model checking algorithm that can verify a property, produce a counterexample when it is deemed invalid, or it can answer that the result is *inconclusive* when the limits provided for the counting fluents are not sufficiently large for the analysis. • CLTSA enhances the LTSA counterexample trace report and the trace animator, providing relevant information regarding counting fluents evaluations.

## 2 INTRODUCING CLTSA

As in LTSA, the system's model is described in CLTSA in terms of the FSP language [9]. In FSP specifications, "`->`" denotes event prefix, "`|`" denotes choice, and conditions can be expressed by means of "`when`" clauses. Processes may be indexed and parameterised, and can be composed in a sequential "`;`" or parallel way "`||`".

One of the main features that LTSA provides is that we can specify temporal properties on the modelled system, and then we can analyse their validity via model checking. LTSA support FLTL (Fluent Linear-Time Temporal Logic) for properties specification. FLTL enriches the traditional LTL logic [7, 8] with propositional fluents. A propositional fluent $Fl = \langle I, T, B \rangle$, is a propositional variable that captures states of the system in terms of activating ($I$) and deactivating ($T$) events, starting with a default value $B$. These fluents can be used as part of the property formula to be verified.

In CLTSA the properties can be expressed in CFLTL [13], an extension of FLTL with counting fluent support. As opposed to the boolean nature of propositional fluents, *counting fluents* represent numerical values that enumerate event occurrences in terms of incrementing, decrementing and resetting events. The syntax of counting fluents declarations in CLTSA is characterised by the following grammar:

⟨*CFluentDef*⟩ ::= ' cfluent ' ⟨*fluent_name*⟩ '='
' <'⟨*incremental_events_set* ⟩ ',' ⟨*decremental_events_set* ⟩ ','
⟨*reset_events_set* ⟩ ' >' ' initially ' ⟨*initial_value* ⟩

Due to their numerical nature, for system's properties specification, counting fluents can be combined to conform a counting expression, i.e. an arithmetical expression that asserts some state of counting fluents. The counting expression can be specified according to the following grammar:

$\epsilon$ ::= ⟨*expr*⟩ ⟨*rel_op* ⟩ ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*value*⟩ | ' (' ⟨*expr*⟩ ' )' | ⟨*expr*⟩ ⟨*arith_op* ⟩ ⟨*expr*⟩

⟨*value*⟩ ::= ⟨*intValue* ⟩ | ⟨*countingFluent* ⟩

⟨*rel_op* ⟩ ::= == | != | < | <= | >= | >

⟨*arith_op* ⟩ ::= + | − | $*$ | / | %

As an example, let us consider the Single Lane Bridge Problem (SLB), a modelling problem introduced in [9], in this case with an additional constraint. Besides the fact that, due to the bridge's width, cars circulating in different directions at the same time 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.

To address this system analysis, as depicted in the Fig. 1, in the CLTSA editor, the counting fluent *CARS_ON_BRIDGE* is declared to keep track of the number of cars (red or blue) on the bridge. This value is initially `0`, is *incremented* at each occurrence of an `enter` (red or blue car) 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 quite natural way, as follows:

`assert` *CAPACITY_SAFE* = [](*CARS_ON_BRIDGE* <= `C`)

The user can find this and other case studies, mostly presented in [13], in the File ≫ Examples ≫ CountingFluents ≫ *case* menu.

The user can perform verification of the properties by selecting them from the Check ≫ *property* menu. Due to the arithmetic nature of the counting fluents, and their potential infinite state representation, some limits to counting fluents possible values must be provided, namely the lower (minimum) and upper (maximum) values that they can take during any system execution. In case of missing limits declarations, as shown in Fig. 1, a window will ask for them. These limits can be applied by means of the `apply` declaration, using the following syntax:

⟨*CFluentDef*⟩ ' apply' ( ⟨*limit_name*⟩ | ⟨*CFluentLimitDef*⟩ )

' limit ' ⟨*limit_name*⟩ '=' ⟨*CFluentLimitDef*⟩

⟨*CFluentLimitDef*⟩ ::= ('[' | ' (') ⟨*min_value*⟩'..' ⟨*max_value*⟩ (']' | ' )')

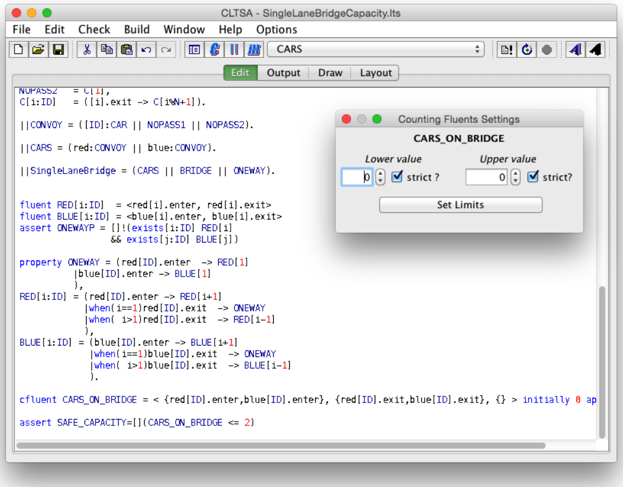
where *brackets* and *parentheses* are used to indicate the *strict* and *non-strict* limits, respectively. Notice that the syntax allows one to define generic limits, with a name, to be applied in one or more counting fluent definitions.

The distinction between the *strict* and *non-strict* limits lies in the behaviour that our model checking approach adopts when a counting fluent has reached its maximum (resp. minimum) value and some incrementing (resp. decrementing) event takes place.

When a *strict limit* is exceeded, the counting fluent value remains as is, on the maximum (resp. minimum) value, and the analysis goes on. On the other hand, when a *non-strict* limit is exceeded, i.e., a *fluent overflow state* has been reached, the current trace is discarded by our model checking algorithm. This procedure guarantees that, if the tool finds a counterexample, that trace never reaches an overflow scenario and the property is reported as *invalid.*

However, if no counterexample was found, but some overflow trace has been explored, then the result will be reported as *inconclusive*, in the sense that the property cannot be deemed valid nor invalid. This situation can take place when the limits defined for counting fluents are not large enough to produce a fully concrete counterexample. On the other hand, if no counterexample has been found, and no overflow trace has been explored, then our approach can guarantee the *validity* of the property being analysed.

Fig. 2 shows an example of an invalid property for which a counterexample was found. The original output of LTSA was modified in order to report the information corresponding to counting fluents along (counterexample) traces.


CLTSA - SingleLaneBridgeCapacity.lts
File Edit Check Build Window Help Options
CARS
Edit Output Draw Layout
Counting Fluents Settings

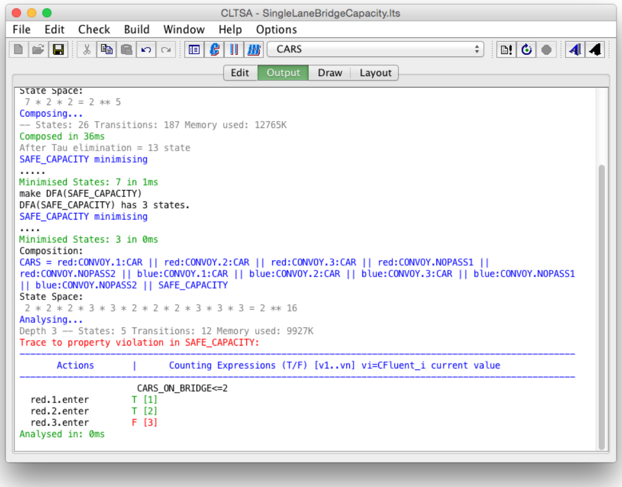
```
NOPASS2   = C[1],
C[i:ID]   = ([i].exit -> C[i%N+1]).

||CONVOY = ([ID]:CAR || NOPASS1 || NOPASS2).

||CARS = (red:CONVOY || blue:CONVOY).

||SingleLaneBridge = (CARS || BRIDGE || ONEWAY).


fluent RED[i:ID]  = <red[i].enter, red[i].exit>
fluent BLUE[i:ID] = <blue[i].enter, blue[i].exit>
assert ONEWAYP = []!(exists[i:ID] RED[i]
                  && exists[j:ID] BLUE[j])

property ONEWAY = (red[ID].enter  -> RED[1]
            |blue[ID].enter -> BLUE[1]
            ),
RED[i:ID] = (red[ID].enter -> RED[i+1]
             |when(i==1)red[ID].exit  -> ONEWAY
             |when( i>1)red[ID].exit  -> RED[i-1]
             ),
BLUE[i:ID] = (blue[ID].enter -> BLUE[i+1]
              |when(i==1)blue[ID].exit  -> ONEWAY
              |when( i>1)blue[ID].exit  -> BLUE[i-1]
              ).

cflunt CARS_ON_BRIDGE = < {red[ID].enter,blue[ID].enter}, {red[ID].exit,blue[ID].exit}, {} > initially 0 ap

assert SAFE_CAPACITY=[](CARS_ON_BRIDGE <= 2)
```

CARS_ON_BRIDGE

| Lower value | | Upper value | |
|---|---|---|---|
| 0 | ☑ strict ? | 0 | ☑ strict? |

Set Limits

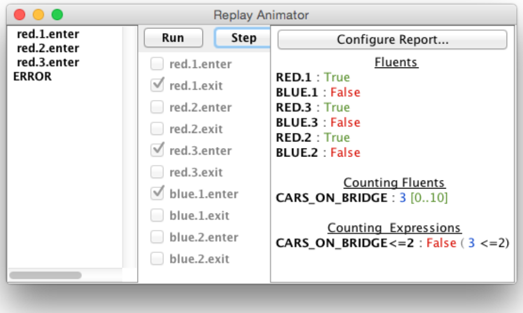
Figure 1: Editor and C.Fluent limits configurator


CLTSA - SingleLaneBridgeCapacity.lts
File Edit Check Build Window Help Options
CARS
Edit Output Draw Layout

```
State Space:
 7 * 2 * 2 = 2 ** 5
Composing...
-- States: 26 Transitions: 187 Memory used: 12765K
Composed in 36ms
After Tau elimination = 13 state
SAFE_CAPACITY minimising
.....
Minimised States: 7 in 1ms
make DFA(SAFE_CAPACITY)
DFA(SAFE_CAPACITY) has 3 states.
SAFE_CAPACITY minimising
....
Minimised States: 3 in 0ms
Composition:
CARS = red:CONVOY.1:CAR || red:CONVOY.2:CAR || red:CONVOY.3:CAR || red:CONVOY.NOPASS1 ||
red:CONVOY.NOPASS2 || blue:CONVOY.1:CAR || blue:CONVOY.2:CAR || blue:CONVOY.3:CAR || blue:CONVOY.NOPASS1
|| blue:CONVOY.NOPASS2 || SAFE_CAPACITY
State Space:
 2 * 2 * 2 * 3 * 3 * 2 * 2 * 2 * 3 * 3 * 3 = 2 ** 16
Analysing...
Depth 3 -- States: 5 Transitions: 12 Memory used: 9927K
Trace to property violation in SAFE_CAPACITY:
-------------------------------------------------------------------------------------------------
      Actions       |      Counting Expressions (T/F) [v1..vn] vi=CFluent_i current value
-------------------------------------------------------------------------------------------------
                     CARS_ON_BRIDGE<=2
  red.1.enter        T [1]
  red.2.enter        T [2]
  red.3.enter        F [3]
Analysed in: 0ms
```

Figure 2: Results of a property check

Another useful feature of the tool is the animator. It provides a window which can simulate the system execution by selecting the enabled events on each step (Check 〉Run 〉*system*). Usually, the animator is very useful for reproducing counterexample traces. CLTSA incorporates a fluents report (see Fig. 3) which shows the values of propositional and counting fluents, as well as the counting expressions, in each step along the trace being animated.



```
red.1.enter
red.2.enter
red.3.enter
ERROR
```

□ red.1.enter
☑ red.1.exit
□ red.2.enter
□ red.2.exit
☑ red.3.enter
□ red.3.exit
☑ blue.1.enter
□ blue.1.exit
□ blue.2.enter
□ blue.2.exit

Fluents

RED.1 : True
BLUE.1 : False
RED.3 : True
BLUE.3 : False
RED.2 : True
BLUE.2 : False

Counting Fluents

CARS_ON_BRIDGE : 3 [0..10]

Counting Expressions

CARS_ON_BRIDGE<=2 : False ( 3 <=2)

Figure 3: Animator window

## 3 ARCHITECTURAL OVERVIEW

In order to describe the implementation of CLTSA, let us first consider an overview of the LTSA model checking process shown in Fig. 4. Basically, the technique consists of checking the emptiness of the synchronous product between the system model $\mathcal{M}$ and the formula negation $\neg\varphi$. In order to use propositional fluents in the formula specification, as proposed in [1], LTSA generates a *fluent automata* for each of them. Intuitively, a *fluent automata* is an automata that consists of two states, representing the truth values of the fluent (*true* or *false*), and a set of transitions labelled with the activating and deactivating events, according the propositional fluent's definition. Finally the product with a *synchronizer automata* induces that for each step that the system takes the automaton corresponding to the fluent updates its state accordingly.

For each step of this process, we highlight (black circled numbers) the modifications introduced in the development of CLTSA.

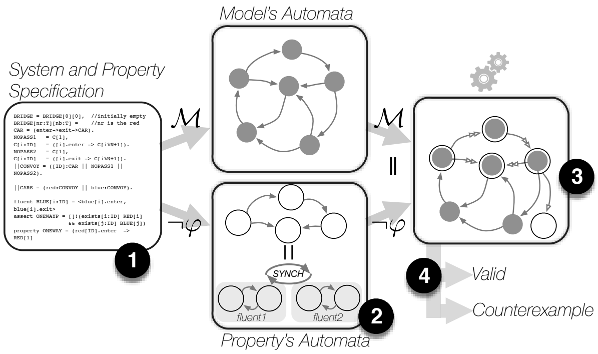


Figure 4: Architectural Overview

❶ In order to use counting fluents in our specifications, we updated LTSA's *lexer* and *parser* to support the following constructions, whose syntax were presented in Sec.2: *limits* definitions, *counting fluents* definitions and *counting expressions* as part of LTL formulas.

❷ Similar to the approach proposed in [1], for each counting expression present in the formula to be verified, our model checking approach generates a *counting automata* that captures the truth value of the corresponding counting expression. As a simple example, consider that we have a counting fluent F, defined as: F =< {a}, {b}, {c} > *initially* 0. Moreover, suppose that we have the counting expression F <= 1 ($\alpha$ in the figure) in the specified property. If we select [0..2] as the *strict* limits for counting fluent F, then Fig. 5 shows the counting automata that CLTSA generates for the counting expression F <= 1.

In case of *non-strict* limits, we add an *overflow* state which is reached through an incrementing (decrementing) event from the

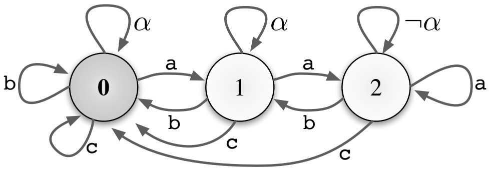


**Figure 5: Counting automata for F <=1.**

maximum (minimum) value state of the counting automata. The *overflow* state works as a *sink* state, in the sense that once this state is reached, then every event that takes place associated to the counting fluent, will self-transition to the *overflow* state. Notice that this state is not a state of acceptance or denial of the corresponding expression value; from this state only an *overflow* situation can be reported.

3 As mentioned before, in the presence of non-strict bounds, our approach can return an inconclusive result. To address this situation, we modify the model checking algorithms present in LTSA. LTSA provides different algorithms for safety and liveness formulas, since the shape of counterexamples will be different in each case.

**Safety properties** Safety properties express that "bad things" will never happen. A counterexample for this kind of property is a *finite trace*. After the composition of the automata is generated, the model checking algorithm looks for a trace that leads us to the ERROR state, i.e., a counterexample that violates de property. To tackle the overflow situations in presence of non-strict bounds, we update the original algorithm by checking that no *overflow* state appear in a counterexample trace. Finally, as mentioned before, for these scenarios we distinguish between these three possible cases with their corresponding results: i) *valid*, when no trace to an ERROR state was found; ii) *invalid*, when a trace to an *ERROR* state was found and no *overflow* state appears in the trace; iii) *inconclusive*, when no counterexample was found, but some *overflow* state was explored.

**Liveness properties** This kind of property expresses that "good things" will eventually happen. A counterexample for this kind of property will be an infinite trace, named a *lasso trace*: a trace conformed by a *prefix* and a *loop*-part, in which a set of events are repeated within a cycle and some of them are undesired. For this kind of properties, LSTA searches for *strongly connected components* (SCC) in which the property to be analysed does not hold.

In a similar way that for safety properties, to tackle *overflow* situations, we update the algorithm by analysing the SCC found in the verification process to distinguish between three possible results: i) *valid*, when no SCC was found; ii) *invalid*, when an SCC was found and it does not contain an *overflow* event; iii) *inconclusive*, if no SCC was found, but some *overflow* state was explored.

4 To enhance the report for the model checking process, we updated the output taking into account the possible *inconclusive* outcome. In addition, in case of invalid properties, i.e., when a counterexample is found, we update the report by providing useful information regarding the value of propositional fluents, counting fluents, and counting expressions at each step of the trace.

# 4 REMARKS

This tool demonstration paper introduced CLTSA, an extension of LTSA with counting fluent temporal logic support. CLTSA allows one to specify and verify LTL properties over reactive systems, providing us with an intuitive and nature way to capture properties related to the number of times that certain system events occur. Due to the potentially infinite size of counting fluents, the user is required to introduce limits for each counting fluent, in order to make the model finite. Moreover, in order not to oversimplify system models and properties regarding counting fluents, different kinds of limits are allowed, allowing for the model checking process to return a third possible result: *inconclusive*.

In comparison with the previous prototype presented in [13], CLTSA incorporates an automata based representation for counting expressions, instead of a monitoring process automatically generated to instrument the model under analysis. In addition, CLTSA enriches the language of the counting expressions, by supporting expressions with an arbitrary number of counting fluents and a wider range of arithmetical operators. In order to produce a more user friendly report of the results, CLTSA also updates the original LTSA trace report with the counting expression status present in the formula to be analysed. Also it incorporates to the Animator the status of fluents, counting fluents and counting expression values at each step of the animation. In terms of efficiency and scalability, it is important to remark that CLTSA was able to efficiently handle all the case studies addressed in [13]. Moreover, the CLTSA's model checking algorithm outperforms that presented in [13], mainly in those cases where the property to be analysed is complex, like liveness properties. This is because the automated instrumentation generated by [13] adds many additional events to the system in order to capture the counting property, producing a considerable increase in the state space required by the formulas and models. Contrary to that approach, in the new automata based implementation, the increase in complexity only affects the automata of the formula, which grows with respect to the range of selected limits for counting fluents. Finally, it is important to remark that CLTSA has shown a good performance for both verifying the validity of properties and generating counterexamples.

Several extensions to LSTA were proposed, for instance [2, 6, 14]. In particular, in [6] an extensive set of LTS layout capabilities was provided for LTSA, which contributes with different layout algorithms, that allow us to manually edit the visualisation graph, navigate from state to state, etc. We would like to thank Cédric Delforge and Charles Pecheur, who kindly allowed us to incorporate these LTS layout features into CLTSA.

Some CLTSA features currently being developed are: i) Counting Fluent indexing and Counting Fluent array arithmetical operations such as array summation. ii) Conditional Counting Fluents: to relate a counting fluent with some propositional fluent $C$, in such a way that the counting fluent values can be updated only when the propositional fluent $C$ is true.

The tool, all the case studies, and a description of how to reproduce the experiments, can be found in http://dc.exa.unrc.edu.ar/tools/cltsa.

## REFERENCES

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

[2] N. D'Ippolito, D. Fischbein, M. Chechik, S. Uchitel, *MTSA: The Modal Transition System Analyser*, 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2008, L'Aquila, Italy, 2008.

[3] 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.

[4] H. Foster, S. Uchitel, J. Magee, J. Kramer, *LTSA-WS: a tool for model-based verification of web service compositions and choreography*, 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, 2006.

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

[6] C. Delforge, C. Pecheur *http://lvl.info.ucl.ac.be/Tools/LTSADelforge*

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

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

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

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

[11] G. Regis, N. Ricci, N. Aguirre, T. S. E. Maibaum, *Specifying and Verifying Declarative Fluent Temporal Logic Properties of Workflows*, Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, 2012.

[12] G. Regis, F. Villar, N. Ricci, *Fluent Logic Workflow Analyser: A Tool for The Verification of Workflow Properties*, Proceedings First Latin American Workshop on Formal Methods, LAFM 2013, Buenos Aires, Argentina, 2013.

[13] G. Regis, R. Degiovanni, N. D'Ippolito, N. Aguirre, *Specifying Event-Based Systems with a Counting Fluent Temporal Logic*, 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Vol. 1, 2015.

[14] P. Rodrigues, E. Lupu, J. Kramer, *LTSA-PCA: tool support for compositional reliability analysis*, 36th International Conference on Software Engineering, ICSE 2014, Companion Proceedings, Hyderabad, India, 2014.