CLTSA: Labelled Transition System Analyser with Counting Fluent support

Germán Regis*, Renzo Degiovanni*†, Nazareno Aguirre*†
*Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Argentina
†Consejo Nacional de Investigaciones Científicas y Técnicas (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 *fluents* by the related concept of *counting fluent*. While fluents allow us to capture propositional states of the behaviour of a reactive system, counting fluents are numerical values, that enumerate event occurrences of them.**

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 in the tool website http://countingfluents.weebly.com. A video highlighting the main features of CLTSA can be found at https://youtu.be/DDLGBsNBazQ.

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 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 [3], CCS [8] and FSP [7], 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 [9]. 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 this paper we present CLTSA, a tool that extends LTSA [7][1] with support for counting fluents. Given an FSP model of a reactive system, CLTSA allows one to specify counting fluents that monitor the behaviour of the system and use them as part of counting expressions for specifying properties. Moreover, the tool can also model check these properties. Since LTL extended with counting fluents is an undecidable formalism [9], our model checking procedure needs to impose bounds on counting fluents values, leading to a sound but inherently incomplete model checking approach. CLTSA supports different kinds of bounds, and a rich language to define these so that the analyses are not over-restricted.

Contribution. Our tool support:

- Counting fluents definition in terms of system events.
- Specification of LTL properties that involve counting fluents, that can be combined by a wide range of arithmetical expressions.
- Definition of different kind of limits for counting fluents, required by our model checking approach.
- An automated model checking algorithm that, not only can verify a property or produce a counterexample when it is invalid, in addition it can answer that the result is *inconclusive* because the limits provided for the counting fluents are not large enough.
- Enhance of the counterexample trace report and the trace animator, integrated to the tool.

II. Introducing CLTSA

Behaviour models are described in CLTSA in the same way these are described in LTSA, i.e., in the FSP language [7]. 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 ("`||`").

Given a system model $\mathcal{M}$, we can specify properties to be analysed via model checking. As opposed to LTSA, properties in CLTSA are expressed in CFLTL [9], an extension of FLTL with support for counting fluent expressions. FLTL enriches the LTL logic [5], [6] with fluents. A fluent Fl, defined as

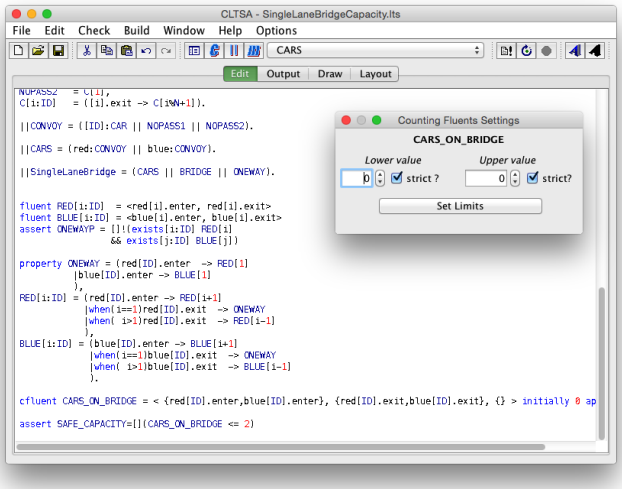

Fig. 1: Editor and C.Fluent limits configurator.

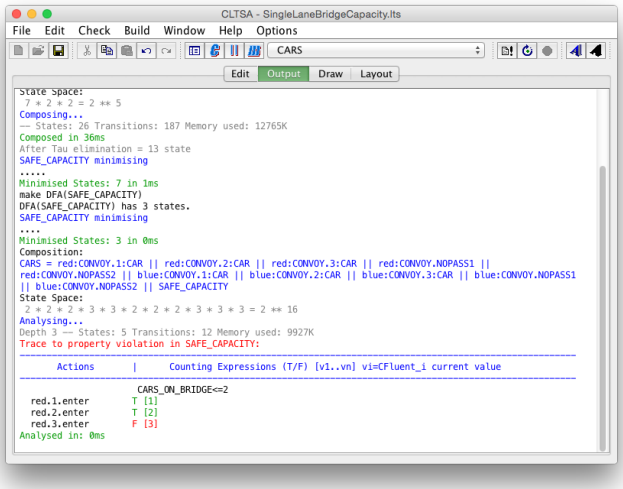

Fig. 2: Property checking results.

$\langle I, T, B \rangle$, 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.

As we mentioned, CLTSA incorporates *counting fluents*, which as opposed to the boolean nature of fluents, represent numerical values that enumerate event occurrences in terms of incrementing, decrementing and resetting events. The syntax of counting fluents declaration in CLTSA is characterised by the following grammar:

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

Counting fluents can be combined to conform a counting expression, i.e. an arithmetical expression that assert about some state of fluents. The counting expression can be specified with following syntax:

ϵ ::= ⟨*expr*⟩ ⟨*rel_op*⟩ ⟨*expr*⟩
⟨*expr*⟩ ::= ⟨*value*⟩ | '(' ⟨*expr*⟩ ')' | ⟨*expr*⟩ ⟨*arith_op* ⟩ ⟨*expr*⟩
⟨*value*⟩ ::= ⟨*intValue* ⟩ | ⟨*countingFluent* ⟩
⟨*rel_op*⟩ ::= == | != | < | <= | >= | >
⟨*arith_op*⟩ ::= + | − | ∗ | / | %

These counting expression can be used in the properties to perform a system analysis. As example, let us consider the Single Lane Bridge Problem (SLB), a modelling problem introduced in [7] (cf. Section 7.2 therein) with an additional constraint. 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.

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 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` (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 more natural way, as follows:

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

The user can find this and other case studies presented in [9] 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 fluent and their potential infinite state representation, some limits to counting fluents must be provided, namely the lower (minimum) and upper (maximum) values that they can take during the 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 following the 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* limit, respectively. Note that the syntax allows 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 yields in the 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 take 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, a *fluent overflowed state* has been reached, so the current trace is discarded by our model checking approach. Then, when the tool produces a counterexample, such a trace guarantees that no counting fluent exceeds its corresponding limits. However, if no counterexample has been found within the limits provided, but a *fluent overflowed* state has been reached, then the result is inconclusive, in the sense

that we cannot inferred that the property is valid, due to the limits may be not sufficient to produce a counterexample. On the other hand, if no counterexample has been generated and no *fluent overflowed* state has been reached, then our approach can guarantee the validity of the property of interested.

Summarising, the output of a property verification, in presence of counting fluents, will be one of the following cases:

- *Invalid*: A counterexample was found (without fluent-overflowed states).
- *Inconclusive*: No counterexamples was found within the limits provided, but a fluent-overflowed state was detected.
- *Valid*: No counterexample was found and no fluent-overflowed state was detected.

The 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.

Another useful feature of the tool is the animator . It provides a window which can simulate the system execution by selecting the enable 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 fluent, as well as the counting expressions, in each step along the trace being animated.

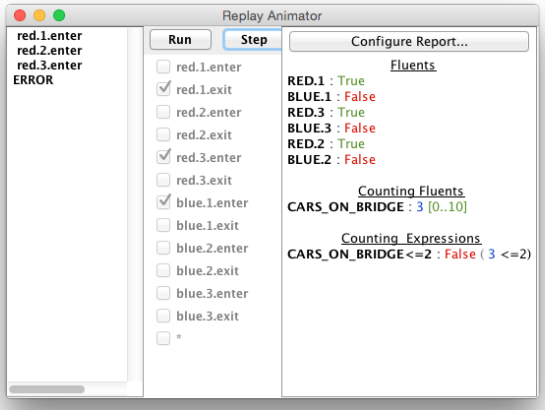

Fig. 3: Animator window.

III. Architectural Overview

CTLA is an extension of the LTSA tool [1] that incorporates counting fluents. Fig.4 shows an overview of the LTSA model checking process, that we later explain how it was modified to support counting fluents analysis. Basically, given an FSP model $\mathcal{M}$ and a FLTL formula φ, LTSA generates an automata for the model $\mathcal{M}$, a *fluent automata* for each fluent definition, and a Buchi automata characterising the negation of the formula (i.e., $\neg\varphi$), and checks the emptiness of the synchronous product between these automatons. Intuitively, a *fluent automata* is an automata that consist of two states, representing the truth values of the fluent (*true* and *false*), and a set of transitions labelled with the activating and deactivating events, according the fluent's definition. LTSA adds two particular self-transitions to these two states, in order to distinguish in which of them the fluent value is true or false.

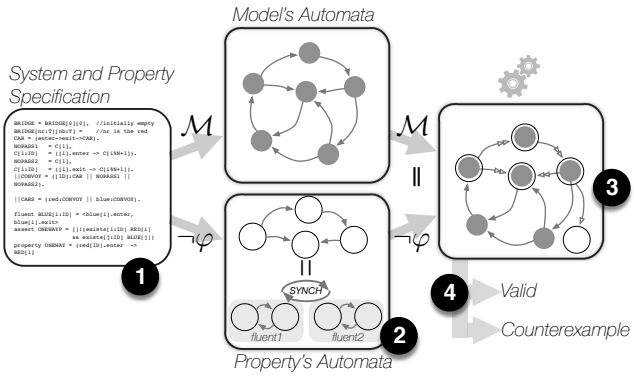

Fig. 4: Architectural Overview

1 In order use counting fluents in our specifications, we update the *lexer* and *parser* to support the following constructions, whose syntax was presented in Sec.II:

- *limits* definitions,
- *counting fluents* definitions, and
- *counting expressions* as part of LTL formulas.

2 Following the approach proposed in [1], for each counting expression present in the formula to verify, our model checking approach generates a *counting automata* that captures the truth value of the corresponding counting expression. As example, the Fig.5 depicts the automata corresponding to the counting expression ϵ of the form $\kappa \leq x$ with a counting fluent defined by $\kappa =< I, D, R > initially\ l+1$, where $[l, u]$ is the κ's *strict* limit, and $l+1 < x < u-1$.

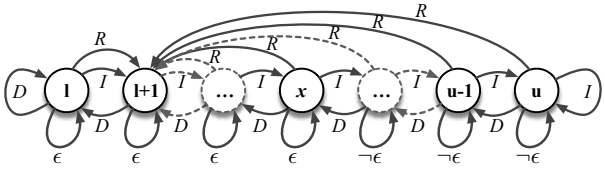

Fig. 5: Counting automata for the counting expression ϵ.

In case of *non-strict* limits, we add an *overflowed* state which is reached trough a incrementing (decrementing) event from the maximum (minimum) value state of the counting automata. The *overflowed* state works as a *sink* state, in the sense that once this state is reached, then every event that take place associated to the counting fluent, will self-transition to the *overflowed* state. Note that this state is not a state of acceptance or denial of the corresponding expression value, from this state only an *overflowed* situation can be reported.

3 As mentioned before, in presence of non-strict bounds, our approach can return an inconclusive result. To address this

situation, we modify the model checking algorithm present in LTSA. LTSA implements its model checking approach by distinguishing safety from liveness formula, since the shape of counterexamples will be different in these cases.

Safety properties Safety properties express that "bad things" will never happen. A counterexample for this kind of properties a finite trace. After composing all of the automatons generated by the model checking approach with the Buchi automata for the negation of the property, our tool proceeds to look for a trace that lead us to the `ERROR` state.

For this process we update the LTSA algorithm by checking that no *overflowed* state appear in a counterexample trace. In this scenario we distinguish between these three possible cases with the corresponding result:

- If no trace to an `ERROR` state was found, then the property will be reported as *valid*.
- If a trace to an *ERROR* state was found and no *overflowed* state appear in the trace, then the property will be reported as *no valid*, and the trace will be returned as counterexample.
- If each trace that leads to an *ERROR* state passes through an *overflow* state, then the property will be reported as *inconclusive*, and larger limits will be required.

Liveness properties This kind of properties express that "good things" will eventually happen. A counterexample for this kind of properties will be a infinite trace, named a *lasso trace*: a trace conformed by two parts, namely, a *prefix* and a *loop*-part, in winch a set of events are repeated into a cycle, where some undesired event occurs. For this kind of properties, LSTA search for *strongly connected components*(SCC) in which the property to be analyse does not hold.

In a similar way that for safety properties, we update this algorithm in order to detect *overflowed* states in the search process, with the following results:

- If no SCC was found, then the property will be reported as *valid*.
- If a SCC was found and no contains some *overflowed* event, then the property will be reported as *no valid*, and the trace will be returned as counterexample.
- If every SCC found contains some *overflowed* event, then the property will be reported as *inconclusive*, and larger limits will be required.

4 To enhance the report for the model checking process, we update the output of the analysis report taken into account the *inconclusive* possible outcome. In addition, in case of no-valid properties, i.e., when a counterexample is found, we update the report by stating the value of propositional fluent, counting fluents, and counting expressions at each step of the trace.

IV. Remarks and Future Works

In this demo, we presented CLTA, an extension of LTSA with counting fluent temporal logic support. CLTA allows one to specify LTL properties on reactive systems that incorporates counting fluents, providing us an intuitive and nature way to capture properties related to the number of times that certain system events occurs. These LTL properties, equipped with counting fluents, can be analysed by the model checking approach that CLTSA provides. Due to the potential infinite size nature of counting fluents, the user is required to introduce limits for each counting fluent, in order to make the model finite. Thus, CLTSA has now three possible outcomes: the property is valid, the property in invalid and a counterexample was generated, and the result is *inconclusive* and larger limits are required.

In order to produce a more user friendly report of the results, CLTA 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 [4], an extensive LTS layout capabilities was provided for LTSA. It offers some different layout algorithms, that allow us to manually edit the graph, navigate from state to state and more. Thank to the Cédric Delforge and Charles Pecheur collaboration, we incorporate to CLTSA these LTS layout features.

Some features which are under development are: i) Counting Fluent indexation and Counting Fluent array's arithmetical operations such as SUM (summation quantification). ii) Conditional Counting Fluents: the idea is to relate a counting fluent with some propositional fluent C, so as to the counting fluent value can be updated only when the propositional fluent C is true; otherwise, i.e., when C is false, the counting fluent value remains frozen.

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] 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.

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

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

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

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

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

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

[9] 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.