Simulation under arbitrary temporal logic constraints

Julien Brunel David Chemouil Alcino Cunha Nuno Macedo
ONERA DTIS and Université fédérale de Toulouse, France INESC TEC and Universidade do Minho, Portugal

Most model checkers provide a useful simulation mode, that allows users to explore the set of possible behaviours by interactively picking at each state which event to execute next. Traditionally this simulation mode can not take into consideration additional temporal logic constraints, such as arbitrary fairness restrictions, substantially reducing its usability for debugging the modelled system behaviour. Similarly, when a specification is false, even if all its counter-examples combined also form a set of behaviours, most model checkers only present one of them to the user, providing little or no mechanism to explore alternatives. In this paper, we present a simple on-the-fly verification technique to allow the user to explore the behaviours that satisfy an arbitrary temporal logic specification, with an interactive process akin to simulation. This technique enables a unified interface for simulating the modelled system and exploring its counter-examples. The technique is formalised in the framework of state/event linear temporal logic and a proof of concept was implemented in an event-based variant of the Electrum framework.

1 Introduction

Model checking is one of the most successful techniques for analysing systems, largely due to the ability to automatically verify whether a temporal logic specification holds in a model of a system. Model validation and debugging is essential when analysing a system, and most model checkers provide a simulation mode where the user can explore alternative system traces by choosing how to proceed with the exploration. With most tools, it is possible to choose one of the possible successor states randomly. Additionally, in order to provide a finer control and speed up the debugging process, many tools also allow the user to interactively pick which event to execute next (if the modelling language has some notion of event/action) and/or one of the next possible states (to support the exploration of non-deterministic events, both features must be provided). These simulation modes are quite intuitive and can even be used by problem domain experts unfamiliar with model checking to help validate the model.

Unfortunately this simulation mode only takes into account the system model, traditionally specified by some sort of transition system or a set of events. However, in some situations it would be extremely helpful to perform such simulation under additional constraints, for example to assess the impact of imposing arbitrary fairness constraints. Such constraints reduce the set of valid behaviours and simulation could help the user validate and better understand their impact (which is not always trivial to infer). Similarly, when model checking a given temporal logic property it could be very useful to explore the set of behaviours that falsify it (its set of counter-examples) with a similar simulation technique. Currently most model checkers display a single counter-example when a property is false. As a consequence, the user often inspects the (lone) counter-example to locate the possible source of the problem, changes the model (or specification) to address it, only for the model checker to reveal a different counter-example to the same property. The ability to explore distinct counter-examples at once could allow the user to identify a more general fix, thus tightening the check / analyse / fix loop and making the overall model checking process more efficient.

In this paper we propose a simulation technique that explores the set of the behaviours that satisfy (or falsify) an arbitrary temporal logic specification. At any point the user can focus on a particular state of

Submitted to:
F-IDE 2019

a trace, see which alternative events enable the same trace prefix to be extended into a complete valid behaviour (another infinite trace satisfying the property), and follow any of those to proceed with the exploration. While traditional simulation is rather easy to implement efficiently for any model resembling a transition system, it is unclear how to do so when additional constraints are imposed. This paper explores the viability of a rather naïve on-the-fly technique: when a state is focused, multiple queries to the model checker are run in the background to determine which events can be further explored, while still preserving the same trace prefix. To tame the complexity in models with many events (or parametrised ones), type categorisation is supported: the user first focuses on a specific type and only then iterates over the different events of that type.

This paper is structured as follows. In the next section we very briefly discuss some alternative techniques to explore the set of behaviours that satisfy (or falsify) a given property. In Section 3 we formalise our proposal in the general setting of event/state linear temporal logic. Section 4 presents a prototype implementation of the proposed technique in the Electrum Analyzer [2], the model checker for the Electrum language [8], an extension of Alloy [7] with linear time temporal logic. The goal of this prototype is mainly to show the viability of the approach, namely in terms of user-experience and efficiency. Section 5 wraps-up the paper and presents some ideas for future work.

2 Related work

Some techniques have been proposed to explore of the set of behaviours that satisfy (or falsify) a given property. The simplest ones just provide iteration over such set, by independently displaying one trace at a time. This can be achieved by changing an explicit model checking engine to resume search after finding one counter-example trace, or, in the case of a SAT-based symbolic bounded model checker, by incrementally adding new clauses that exclude exactly the previous trace, as implemented in the Electrum Analyzer [2] developed by the authors. The problem is that this frequently keeps yielding traces that are just slight variations of each other and, since the full set of behaviours is usually too big to be enumerated, finding interesting variations may prove infeasible. To alleviate this problem, for specific modelling languages it is possible to define reasonable equivalence classes on traces (e.g., traces that follow the same control-flow path are deemed equivalent), and implement iteration by restarting the model checker with a modified property that conjoins the original one with a formula excluding all traces in the class of the previous counter-example [6, 3].

Problem domain expertise, namely some kind of user input, could lead to more effective exploration. While in the above techniques user interaction is limited to just asking for the next trace, in [5], by running multiple queries to the model checker, a proof tree of a CTL property is inferred to "explain" a counter-example trace, with which the user can interact to ask for new counter-examples. Possible interactions include asking for alternative proofs (e.g., in a disjunction node), or guiding the search to explore different parts of the model (e.g., in $\texttt{EX}\ \phi$ nodes, by choosing the next ϕ-satisfying state). However, this approach requires substantial knowledge of the underlying proof system for CTL and it is not clear how it can be generalised to support LTL and fairness constraints.

3 Formalisation

Most systems incorporate both the notion of states and events. *State/event linear temporal logic* (SE-LTL) was proposed to allow a more concise and intuitive specification in these cases [4]. The semantics of a formula in this logic is defined over a *labelled Kripke structure* (LKS), a tuple $(S, I, P, \mathscr{L}, T, \Sigma, \mathscr{E})$ where S

is a finite set of states, $I \subseteq S$ the set of initial states, P a finite set of atomic propositions, $\mathscr{L} : S \to 2^P$ a state labelling function, $T \subseteq S \times S$ a transition relation, Σ a finite set of events, and $\mathscr{E} : T \to 2^\Sigma \setminus \{\emptyset\}$ a transition labelling function. The transition relation is assumed to be total, so every state has at least one successor. To enable a more efficient exploration, events are categorized with a function $\mathscr{T} : \Sigma \to \Upsilon$ that assigns a type to each event. This categorization is natural in many models, namely those with parametrised events. A *path* $\pi = \langle s_0, a_0, s_1, a_1, \ldots \rangle$ of such a *typed LKS* is an alternating infinite sequence of states and events where $\forall i \cdot (s_i, s_{i+1}) \in T \wedge a_i \in \mathscr{E}(s_i, s_{i+1})$ and $s_0 \in I$.

Given a typed LKS, SE-LTL formulas are defined by the following grammar, where p ranges over P, a over Σ, and t over Υ:

$$\phi ::= p \mid a \mid t \mid \top \mid \neg\phi \mid \phi \wedge \phi \mid \mathtt{X}\,\phi \mid \mathtt{G}\,\phi \mid \mathtt{F}\,\phi \mid \phi\,\mathtt{U}\,\phi$$

Given a path π, the semantics of a formula is the standard one of LTL with the addition that $\pi \models a$ iff a is the first event of π and $\pi \models t$ iff a is the first event of π and $\mathscr{T}(a) = t$. $M \models \phi$ means that ϕ holds in the typed LKS M, that is, for every path π of M we have $\pi \models \phi$. Given a formula ϕ the goal of a model checker is to find a path π such that $\pi \not\models \phi$. We will denote the first such counter-example, if it exists, by $M(\phi)$. Given a path π, $[\pi]_i$ is a formula that exactly characterises the prefix of π up to i, defined as $([s_0] \wedge a_0) \wedge \mathtt{X}([s_1] \wedge a_1) \wedge \ldots \wedge \mathtt{X}^{i-1}([s_{i-1}] \wedge a_{i-1})$, where $\mathtt{X}^i$ is a nesting of i "next" operators and $[s]$ is a formula that fixes the values of the propositions of state s, defined as the conjunction of all propositions appearing in $\mathscr{L}(s)$ and all negated propositions in $P - \mathscr{L}(s)$.

Following [9], our interactive exploration technique is specified by a set of scenario exploration operations. The state of the exploration is a tuple (ϕ, π, i, Φ) where ϕ is the formula being model checked[1], π the current counter-example on display, i the state the user is focused in, and Φ a function mapping each path index to a formula that characterises the set of states and transitions the model checker is allowed to explore at that point. Notation $\Phi \oplus \{i \mathinner{..} j\} \mapsto \psi$ will denote an update on this last function, that maps every index between i and j to ψ, keeping all other indexes intact. When updating a single index i, the notation will be simplified to $\Phi \oplus i \mapsto \psi$.

When first checking a property ϕ this state is initialised as $(\phi, M(\phi), 0, \mathbb{N} \mapsto \top)$. Basic navigation operations can then be used to inspect the counter-example, namely $\bowtie(\phi, \pi, i, \Phi) = (\phi, \pi, i+1, \Phi)$ and $\bowtie\!\!\!\!\!\!\bowtie(\phi, \pi, i, \Phi) = (\phi, \pi, i-1, \Phi)$ (for $i > 0$). At any point i it is possible to ask for a new counter-example that differs only in the outcome of the previous event, a useful operation to explore non-determinism. This operation is defined as $\triangleright(\phi, \pi, i, \Phi) = (\phi, M(\varphi), i, \Phi \oplus (i \mapsto \Phi(i) \wedge \neg[s_i]) \oplus (\{i+1 \mathinner{..}\} \mapsto \top))$, where φ is $\phi \vee \neg([\pi]_i \wedge \mathtt{X}^i(\Phi(i) \wedge \neg[s_i]))$. By repeatedly applying $\triangleright$ all possible outcomes of the previous action will eventually be enumerated (or possible initial states when $i = 0$). Notice how Φ is used to trim a branch of the behaviour tree when this operation is selected, but maintains memory of previously trimmed branches while inspecting a trace with $\bowtie$ and $\bowtie\!\!\!\!\!\!\bowtie$.

Similarly, it is possible to ask for a new counter-example that picks a different next event of the same type. This operation is defined as $\blacktriangleright(\phi, \pi, i, \Phi) = (\phi, M(\varphi), i, \Phi \oplus (i \mapsto \Phi(i) \wedge \neg([s_i] \wedge a_i)) \oplus (\{i+1 \mathinner{..}\} \mapsto \top)$, where φ is $\phi \vee \neg([\pi]_i \wedge \mathtt{X}^i(\Phi(i) \wedge [s_i] \wedge \neg a_i \wedge \mathscr{T}(a_i)))$. Notice how Φ keeps track that the branch starting in $[s_i]$ and labeled with a_i has already been explored. To ask for a new counter-example with a specific type t for the next event, operation $\triangle_t\ (\phi, \pi, i, \Phi) = (\phi, M(\varphi), i, \Phi \oplus \{i+1 \mathinner{..}\} \mapsto \top)$ can be used, where φ is defined as $\phi \vee \neg([\pi]_i \wedge \mathtt{X}^i([s_i] \wedge t))$.

[1] To simplify, in the remaining of the paper we will present the technique in the context of counter-example exploration in model checking, but it can obviously be also used for exploring the valid behaviours of a system with additional arbitrary constraints specified over it, by just running the model checker on the negation of their conjunction and interpreting the resulting set of counter-examples as witnesses of the system's behaviour.

```
1  open util/ordering[Key]
2  sig Key {} sig Room { keys: set Key, var current: one keys } sig Guest { var gkeys: set Key }
3  one sig Desk { var lastKey: Room → lone Key, var occupant: Room → Guest }
4
5  event In[g: Guest, r: Room, k: Key] modifies gkeys, occupant, lastKey {
6     no r.(Desk.occupant) and k = nextKey[r.(Desk.lastKey), r.keys]
7     gkeys' = gkeys + g→k
8     Desk.occupant' = Desk.occupant + r→g
9     Desk.lastKey' = Desk.lastKey ++ r→k }
10 event Out[g: Guest] modifies occupant { ... }
11 event Entry[g: Guest, r: Room, k: Key] modifies current { ... }
12 event Reentry[g: Guest, r: Room, k: Key] { ... }
13 fun nextKey[k: Key, ks: set Key] : set Key { min[nexts[k] & ks] }
14 fact Init { keys in Room lone → Key and no Guest.gkeys and ... }
15
16 assert BadSafety { always { all r: Room, g: Guest, k: Key |
17    (Entry[g,r,k] or Reentry[g,r,k]) and some r.(Desk.occupant) ⇒ g in r.(Desk.occupant) } }
18 check BadSafety for 3 Key, 1 Room, 2 Guest, 10 Time
```

Figure 1: Hotel example in Electrum with events.

4 Implementation

Electrum is an extension of the popular Alloy formal specification language, developed for the analysis of dynamic systems. An Alloy model consists of a set of static signatures and relations (of arbitrary arity). Properties can be specified in an extension of first-order logic: apart from the standard connectives and quantifiers, Alloy supports closures and some derived relational logic connectives, such as composition (`.`) or Cartesian product (→). To make the verification decidable, the user must specify a scope setting the maximum size of all signatures. Counter-examples are depicted graphically with user-customisable themes. In Electrum, signatures and relations can be declared mutable (with keyword `var`) and properties can be specified using linear temporal logic connectives (including past ones) and primed expressions (denoting their value in the next state) in addition to Alloy connectives.

Recently, we added the notion of event to Electrum [1]. Figure 1 presents an example of an Electrum model with events based on a classic Alloy example that specifies a protocol for disposable room key-cards in a hotel. There are 4 events in this model (check-`In`, check-`Out`, `Entry`, and `Reentry`), each specified declaratively with relational logic and primed expressions. The keyword `modifies` is used to fix the frame. The desired safety property is that only guests registered as occupants of a room can indeed enter that room. Unfortunately, that is not the case and the `check BadSafety` command yields a counter-example trace where a guest checks in, enters the room after checking out, a second guest checks in, and the first guest reenters the room afterwards. This is possible because the door lock has not yet been recoded with the new key issued by the front desk. The previous version of the Electrum Analyzer [2] already allowed the user to ask for full alternative counter-example traces, but each one could only be inspected independently (by navigating backward and forward in the states), making it difficult to understand the relationship between the different counter-examples.

The new prototype interface for simulation and counter-example exploration is depicted in Fig. 2, which illustrates precisely the exploration of the above counter-example at $i = 1$. As in the previous version, the user can focus on a particular state of a trace by navigating backward (⏪) and forward (⏩), using the left- and right-arrows in the bottom toolbar. However, two states are now shown side-by-side, allowing the user to better understand what is the effect of an event. In the top toolbar we also depict the trace and which transition is being inspected, and the bottom toolbar in the middle shows the event that

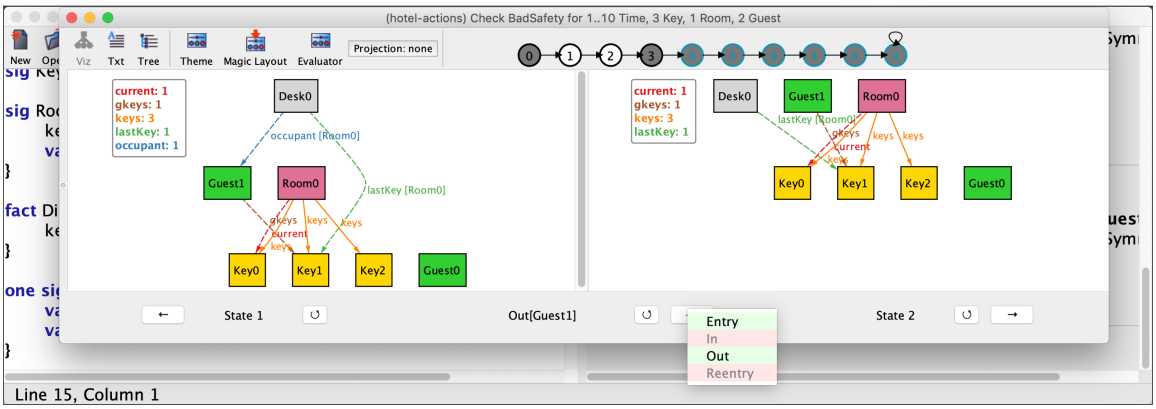

Figure 2: Exploration interface.

triggers the current transition. Following the formalisation in the previous section, the user can choose a different pre- or post-state (the small "reload" buttons under the left- and right-panes, corresponding to operation $\triangleright$), an event of the same type with different parameters (the "reload" button next to the event name, corresponding to the $\blacktriangleright$ operation), or a different event type to execute (the selection button in the bottom toolbar, implementing the $\triangle$ operation). When the user focuses on a state, operation $\triangle$ is dry run on-the-fly to determine which event types are enabled, so that when the event selection button is pressed only the enabled events can be selected (shown with a green background, as opposed to red for the disabled ones). In Fig. 2 we can see that after check-in the only options are for the first guest to check out or enter the room. Unlike in the previous version of the Analyzer, it is now easy to understand that, for the given scope, the check-in of the second guest must necessarily be followed by an entry or reentry of the first guest, and there are no other possibilities to breach safety.

To assess the efficiency of the proposed technique, we measured the required time to determine which $\triangle$ operations are enabled in the different states of the first counter-example returned by the Analyzer. Table 1 shows the results of this preliminary evaluation for different scopes. The first column shows the configuration (number of guests and a list with the number of keys per room), the second the time to compute the first counter-example, and then, for each state i, the total time to compute which event types are enabled, which event was executed (the subscript identifies the guest) and what other event types were enabled. The evaluation was performed with the bounded model checking engine of Electrum (with the Glucose SAT solver), with maximum trace length of 10, in a commodity laptop with a 2.3 GHz Intel Core i5 and 16 GB of RAM. As can be seen, only for $i = 0$ in the last configuration did the solving of all $\triangle$ events take more than 2s, and in most cases it is in the order of a few hundred ms. Since a user typically needs some time to understand a state after focusing, this delay is almost always unnoticed. Also, times tend to decrease as the user advances in the trace: this is to be expected, since a bigger prefix of the trace is fixed, resulting in a smaller search space for the verification engine.

5 Conclusion

This paper presented a simple technique that allows the user to explore the behaviours that satisfy (or falsify) an arbitrary temporal logic specification, with an interactive process akin to simulation. A prototype was implemented in the Electrum Analyzer, and a preliminary evaluation showed its viability in terms of efficiency. In the future we intend to further improve efficiency by testing which events are

C	T	T_0	a_0	T_1	a_1	T_2	a_2	T_3	a_3	T_4	a_4	T_5	a_5	$\cdots$
2[3]	0.07	0.33	$\mathbf{I_1}$	0.20	$\mathbf{O_1}$E	0.18	$\mathbf{I_0}$E	0.22	$\mathbf{E_1}$	0.09	$\mathbf{R_1}$OE	0.09	$\mathbf{R_1}$OE	$\cdots$
2[1,3]	0.06	0.49	$\mathbf{I_1}$	0.30	$\mathbf{E_1}$O	0.27	$\mathbf{O_1}$R	0.23	$\mathbf{I_0}$R	0.26	$\mathbf{R_1}$	0.11	$\mathbf{R_1}$OE	$\cdots$
3[2,3]	0.11	0.75	$\mathbf{I_2}$	0.34	$\mathbf{O_2}$IE	0.39	$\mathbf{I_1}$E	0.35	$\mathbf{E_2}$I	0.06	$\mathbf{R_2}$IOE	0.07	$\mathbf{R_2}$IOE	$\cdots$
3[1,1,4]	0.58	1.24	$\mathbf{I_2}$	0.77	$\mathbf{O_2}$E	0.62	$\mathbf{I_1}$E	0.53	$\mathbf{E_2}$O	0.23	$\mathbf{R_2}$OE	0.20	$\mathbf{R_2}$OE	$\cdots$
4[1,1,6]	1.74	2.30	$\mathbf{I_3}$	1.41	$\mathbf{O_3}$E	1.10	$\mathbf{I_2}$E	0.94	$\mathbf{E_3}$O	0.39	$\mathbf{R_3}$OE	0.33	$\mathbf{R_3}$OE	$\cdots$

Table 1: Performance of the event type enumeration.

enabled in parallel. To show the generality of the technique we intend to apply it to other model checkers, namely develop a counter-example exploration tool for SMV. Finally, we also plan to conduct a more detailed evaluation, focusing not only on efficiency, but also on its effectiveness, namely in helping the user identify truly different counter-examples.

Acknowledgements

This work is financed by the ERDF - European Regional Development Fund - through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 - and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826, and the French Research Agency project FORMEDICIS ANR-16-CE25-0007. The third author was also supported by the FCT sabbatical grant with reference SFRH/BSAB/143106/2018.

References

[1] Julien Brunel, David Chemouil, Alcino Cunha, Thomas Hujsa, Nuno Macedo & Jeanne Tawa (2018): *Proposition of an Action Layer for Electrum*. In: *Proceedings of the 6th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)*, *LNCS* 10817, Springer, pp. 397–402.

[2] Julien Brunel, David Chemouil, Alcino Cunha & Nuno Macedo (2018): *The Electrum Analyzer: Model checking relational first-order temporal specifications*. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, ACM, pp. 884–887.

[3] Kalou Cabrera Castillos, Hélène Waeselynck & Virginie Wiels (2015): *Show Me New Counterexamples: A Path-Based Approach*. In: *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, pp. 1–10.

[4] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina & Nishant Sinha (2004): *State/Event-Based Software Model Checking*. In: *Proceedings of the 4th International Conference on Integrated Formal Methods (iFM)*, *LNCS* 2999, Springer, pp. 128–147.

[5] Marsha Chechik & Arie Gurfinkel (2007): *A framework for counterexample generation and exploration*. *International Journal on Software Tools for Technology Transfer* 9(5–6), pp. 429–445.

[6] Alma L. Juarez Dominguez & Nancy A. Day (2013): *Generating multiple diverse counterexamples for an EFSM*. Technical Report CS-2013-06, University of Waterloo.

[7] Daniel Jackson (2012): *Software Abstractions: Logic, Language, and Analysis*, 2nd edition. MIT.

[8] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha & Denis Kuperberg (2016): *Lightweight specification and analysis of dynamic systems with rich configurations*. In: *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, ACM, pp. 373–383.

[9] Nuno Macedo, Alcino Cunha & Tiago Guimarães (2015): *Exploring Scenario Exploration*. In: *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE)*, *LNCS* 9033, Springer, pp. 301–315.