UML state machines: Fairness Conditions specify the Event Pool *

Harald Fecher[†], Heiko Schmidt[‡] and Jens Schönborn[§]

Abstract

The communication between different instances of UML state machines is handled by using underlying event pools but the UML standard leaves the behavior completely unspecified. Thus, in general, liveness properties cannot be verified. We give semantics of Streett fairness constraints for the event pool and present an algorithm that turns a set of fairness constraints into an abstract event pool. Since common fairness suffers from the drawback that the time until something good happens may be unbounded the presented algorithm allows the modeler to specify such a bound. The resulting abstract event pool can be further refined, justified by an example where a priority scheme on events is introduced.

1 Introduction

The communication between different instances (objects) of UML state machines is handled by using underlying event pools: An object (caller) can call a method of another object (callee), thereby sending an event to the callee. This event is first received by an implicit, usually not modeled event pool of the callee. At a later point in time, the event is provided by the callee's event pool to the callee's state machine and then, triggers transitions or is discarded otherwise. The UML standard [4] leaves the behavior of the event pool completely unspecified. This flexible treatment has the disadvantage that, in general, liveness properties cannot be verified. Therefore, the modeler should be provided with a facility to restrict the event pool's behavior. We propose adding Streett fairness constraints [3] to the state machine. This allows the modeler to specify fairness constraints without modeling the event pool of the state machine, thereby following the philosophy of the UML standard that the event pool should be left as unspecified as possible. Furthermore, as stated in [1, 2], liveness is robust (independent of the granularity of transitions) and simple (abstraction of complicated time bounds), but it suffers from the drawback that the time until something good happens may be unbounded. Thus the modeler should be enabled to specify a bound for the maximum of time until something good must happen.

Example 1 *A model of a pedestrian crossing on a road controlled by a traffic light is depicted on the left of Figure 1. This model ensures mutual exclusion for the use of the road, but not liveness, i.e., that a request for green light will be serviced eventually. Fairness constraints are needed to avoid possible starvation, and for practical application, this fairness should be bounded.*

Contribution. In order to enable reasoning with liveness properties, we extend UML state machines by fairness constraints for the underlying event pools. We give semantics of unbounded (common) and bounded Streett fairness constraints for the event pool, which is not straightforward, since a non-empty event pool must always provide an event. We present an algorithm that turns a set of fairness constraints into an abstract event pool satisfying the fairness constraints by its structure. As a consequence, traces rejected by the fairness constraints no longer occur in the composition of event pool and state machine. The abstract event pool is modeled using a simplified variant of state machines with a straightforward semantics in terms of labeled transition systems. We argue, by means of an example where a priority scheme on the events is added, that this is feasible, because thereby the event pool can be refined later in the modeling process.

*This work is in part financially supported by the DFG project *FE 942/1-1*.

[†]hfecher@doc.ic.ac.uk
[‡]hsc@informatik.uni-kiel.de
[§]jes@informatik.uni-kiel.de

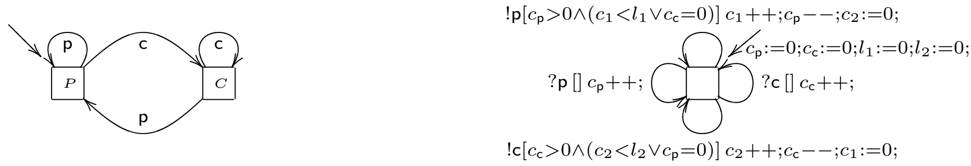

Figure 1: Left: A state machine model of a pedestrian crossing on a road controlled by a traffic light. Pedestrians request green light using a button triggering an event p, an inductor generates events c for cars. The states P and C represent green light for pedestrians resp. cars. These events are sent from the environment via the event pool to the state machine. Right: An abstract model of an event pool guaranteeing fairness between the events p, c in the sense that they cannot be neglected more than l_1-, resp. l_2-, times.

2 State machine and event pool with fairness

Fairness constraints on the state machines specify constraints on the event pool. For the definition of the event pool, we use a variant of state machines in order to enhance readability for software engineers. The semantics of this state machine variant is straightforward. Events sent to the state machine are first immediately received (input enabledness) by the event pool (?e) and will later be sent (provided) to the state machine (!e) via handshake communication. The sender of the event e needs to synchronize with ?e of the event pool and the directive !e of the event pool needs to synchronize with a transition having event e as label, or if no such transition exists, the event is discarded.

2.1 Bounded and unbounded fairness.

We generalize the common notion of fairness, expressed by Streett fairness constraints, such that also finite fairness constraints can be expressed, similar to [1, 2]. A set of fairness constraints of the form (E, F, l), where E and F are sets of events and $l \in \mathbb{N} \cup \{\infty\}$, specifies that at most l events from F may be sent to the state machine, before sending an event from E is required, provided such an action exists in the event pool. A trace t of the event pool, being a sequence of symbols ?e (receive) and !e (send), is accepted by a set of fairness constraints if for each fairness constraint (E, F, l) and for each index k in the trace there are no events from E currently stored in the event pool (i.e., the prefix of t until index k contains equally many occurrences of ?e and !e) or there is no immediately preceding subtrace with more than l occurrences of F send actions (!f with $\mathsf{f} \in F$) without any occurrence of E send actions (!e with $\mathsf{e} \in E$).

Note that bounded fairness specifications can introduce deadlock: Consider the fairness specifications $\{(\{e_1\}, \{e_2, e_3\}, 1), (\{e_3\}, \{e_1, e_2\}, 1)\}$ and the situation, where $!e_2$ has been dispatched and there are occurrences of all events in the event pool. The first tuple requires $!e_1$ to appear before a possible $!e_3$, whereas the second tuple requires $!e_3$ to appear before a possible $!e_1$.

The bound of a fairness constraint can be decreased via refinement. When unbounded fairness constraints are used, i.e. the bounds equal ∞, the resulting event pool is not yet concrete, and therefore the bound has to be decreased via refinement. A concrete event pool will provide, if not empty, exactly one event.

2.2 Transformation

We present the algorithm for generating the event pool from a given set of fairness constraints in Figure 2. The algorithm is divided into two parts: in the first loop on the set of events the basic I/O behavior is modeled and in the second loop on the set of fairness constraints the behavior is further restricted.

Example 2 *In Example 1, pedestrians' requests should not starve cars' requests and vice versa. This is specified by the fairness constraints* $\{(\{\mathsf{c}\}, \{\mathsf{p}\}, l_1), (\{\mathsf{p}\}, \{\mathsf{c}\}, l_2)\}$ *with reasonable values of* l_1, l_2 *which yields an event pool as depicted on the right of Figure 1.*

Define one state s, source and target of all transitions

1. For every event e of the state machine
   a) Define a counter c_e for the number of occurrences of e in the event pool
   b) Define a transition for receiving e labeled $?e[]c_e++;$
   c) Define a transition for sending e labeled $!e[c_e > 0]c_e--;$

2. For every fairness constraint $A_i = (E_i, F_i, l_i)$
   a) Define limit counter c_i
   b) For every $e \in F_i$, add to the transition for sending e
      i. Guard ... $\wedge (c_i < l_i \vee \bigwedge_{e' \in E_i} c_{e'} = 0)$
      ii. Action ...; c_i++;
   c) For every $e \in E_i$ add action $c_i := 0$; to the transition for sending e

Figure 2: The algorithm for generating an abstract event pool from a given set of fairness constraints.

3 Adding priority via refinement

In this section we present an example which shows the feasibility of our approach since it admits further refinement, such as defining priority on events.

We have shown how (bounded) fairness can be realized in the event pool. However, it is also often useful to model priority on events. This can contradict existing fairness constraints, and to solve this, we give fairness constraints higher priority than priority specifications, i.e., a non-prioritized event is handled, in case this is required by a fairness constraint.

Example 3 *We extend the state machine from Example 1 as follows: An emergency state is added, which is requested by an approaching ambulance and turns all lights turn red. This emergency request (*e*) has higher priority than pedestrians' and cars' requests, being handled as soon as it appears. The state machine and its resulting event pool are presented in Figure 3. Here, priority is translated into a hierarchical nesting of state machine states, utilizing the fact that in UML transitions with sources of inner states have priority over transitions with sources of outer states.*

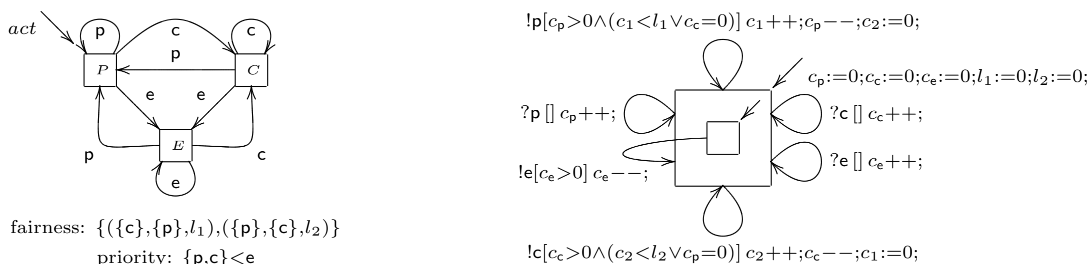

Figure 3: Left: Prioritized pedestrian crossing. A high-priority event e switches into an emergency mode that turns all lights red, issued, e.g., by an ambulance. Right: The resulting abstract event pool.

References

[1] K. Chatterjee and T. A. Henzinger. Finitary winning in omega-regular games. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2006.

[2] N. Dershowitz, D. N. Jayasimha, and S. Park. Bounded fairness. In N. Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *Lecture Notes in Computer Science*, pages 304–317. Springer, 2003.

[3] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.

[4] Object Management Group. *UML Superstructure Specification, v2.1.1 formal/2007-02-03*, 2007.