

# Semantics of Driver Protocol State Machines

Timothy Bourke\* and Leonid Ryzhyk\*

## Abstract

As part of research to develop a high-reliability I/O framework, a statecharts-like notation is being used to describe interactions between the framework/operating system and individual device drivers. Statecharts is a powerful but complicated notation. This document formalizes a simple subset of the notation with orthogonality, hierarchy and state time-outs. Specifically it does not include: history, multi-level transitions, conditions, or actions. The idea is to formalize just enough features to make modelling convenient while retaining a strong intuitive link to underlying automata. There is nothing novel in the formalization, although a more strongly synchronising notion of orthogonality than usual is employed; more like Argos [2] than Statecharts [1].

## 1 Semantics of protocol state machines

Protocol state machines resemble the UML version of Statecharts. This is compatible with the intention to document the driver framework for developers. However, the complexity and non-trivial semantics of Statecharts can complicate analysis and make reasoning difficult. For this reason we restrict the features that may be used and give a formal semantics in terms of simple automata; although we do augment them with watchdog timers. The construction of the semantics borrows from the synchronous language Argos [2] but the purpose and approach differ. Our automata restrict when inputs may occur, i.e. they are not input enabled. The parallel operator is synchronizing but actions are considered one-by-one (serialized). Transitions do not correspond to reactions of a device driver, but rather triggers and call-backs across an interface.

A protocol state machine is a tree. The leaf or external nodes are basic automata. There are two types of internal node: *parallel* nodes whose children run concurrently, and *refinement* nodes that are associated with a basic automaton and whose children are each associated with a unique state of that automaton.

**Definition 1** A basic automaton is a 4-tuple  $(S, i, A, T)$  where  $i \in S$  and  $T \subseteq S \times A \times S$ . We will write  $s \xrightarrow{a}_T s'$  for a triple  $(s, a, s') \in T$ .

**Definition 2** A watched automaton is a 5-tuple  $(S, i, A, \Omega, T)$  where  $i \in S$ ,  $\Omega = (W, watch, value)$ ,  $T \subseteq S \times A \times 2^W \times S$ ,  $watch : S \rightarrow 2^W$ , and  $value : W \rightarrow \mathbb{N}$ . We will write  $s \xrightarrow{a, R}_T s'$  to mean  $(s, a, R, s') \in T$ .

---

\*CSE/NICTA, University of NSW

The  $\Omega$  component of a watched automaton tracks active watchdog timers. The function `watch` maps each state to a set of active timers and `value` maps each timer to a value. Transitions are labelled with a set,  $R$ , of timers to be reset when the labelling action occurs. The timers are intended to detect abnormal behaviour at runtime, they do not influence the set of permitted action sequences. A basic automaton is recovered from a watched automaton by dropping the  $\Omega$  component and the  $R$  component of  $T$ . There is a case for using standard Timed Automata with safety invariants.

The translation from a flat UML state machine into a watched automaton is almost direct. Final states (ringed, solid circles) are simply states that source no further transitions. Conditional junctions are replaced with one transition for each path; labels are concatenated but the directionality marker (? or !) remains as the final character. For example in Figure 2(d) the top-most conditional junction becomes two transitions: `txPacketPull[false]!` and `txPacketPull[true]!`. The set of actions,  $A$ , of a translated flat state machine comprises all the actions labelling transitions and no others.

We do not allow any output actions or internally emitted events, thus the variable manipulations in Figure 2(b) should be seen as macros. Each state may be labelled with a single timing bound (such as  $< 5s$ ), a timer is introduced and included in the reset component of all transitions which exit to a different state. Self-loops do *not* reset the timer of their source state. The `watch` function for states without timers maps to the empty set.

The semantics of concurrent and refined states are defined in terms of two operators on watched automata:  $M_1 \parallel M_2$  combines two concurrent watched automata,  $M_1 \triangleright_s M_2$  refines state  $s$  of  $M_1$  by  $M_2$ . The semantics of a state machine ‘tree’ is the watched automaton built by applying these operators upward from the leaf nodes. Given commutativity and associativity of  $\parallel$ , and forbidding multiple refinements of a single state gives a unique result.

**Definition 3** *Given two watched automata,  $M_1 = (S_1, i_1, A_1, \Omega_1, T_1)$  and  $M_2 = (S_2, i_2, A_2, \Omega_2, T_2)$  where  $S_1 \cap S_2 = \emptyset$ , define a third  $M = M_1 \parallel M_2 = (S, i, A, \Omega, T)$ :*

- $S = S_1 \times S_2$

- $i = (i_1, i_2)$

- $A = A_1 \cup A_2$

- $\Omega = ( \begin{array}{l} W_1 \dot{\cup} W_2, \\ \text{watch}((s, t)) = \text{watch}_1(s) \cup \text{watch}_2(t), \\ \text{value}(w) = \begin{cases} \text{value}_1(w) & \text{if } w \in W_1 \\ \text{value}_2(w) & \text{if } w \in W_2 \end{cases} \end{array} )$

- $T$  is the smallest set defined by:

$$1 \frac{s \xrightarrow{a, R_1}_{T_1} s' \quad a \in A_1 \setminus A_2 \quad t \in T_2}{(s, t) \xrightarrow{a, R_1}_T (s', t)} \quad 2 \frac{t \xrightarrow{a, R_2}_{T_2} t' \quad a \in A_2 \setminus A_1 \quad s \in T_1}{(s, t) \xrightarrow{a, R_2}_T (s, t')}$$

$$3 \frac{s \xrightarrow{a, R_1}_{T_1} s' \quad t \xrightarrow{a, R_2}_{T_2} t' \quad a \in A_1 \cap A_2}{(s, t) \xrightarrow{a, R_1 \cup R_2}_T (s', t')}$$

Importantly, the parallel operator synchronizes those actions common to constituent automata ( $A_1 \cap A_2$ ), such an action is only legal when explicitly allowed by *both*. Note that this is expressly different to the usual definition (see for instance [1, Figures 19 and 20]). No additional constraints are placed on actions allowed by one automata and not mentioned by the other. Timing constraints are maintained orthogonally.

**Definition 4** Given two watched automata,  $M_1 = (S_1, i_1, A_1, \Omega_1, T_1)$  and  $M_2 = (S_2, i_2, A_2, \Omega_2, T_2)$  where  $S_1 \cap S_2 = \emptyset$ , and a distinguished state in the first  $r \in S_1$ , define a third automaton  $M = M_1 \triangleright_r M_2 = (S, i, A, \Omega, T)$ :

- $S = (S_1 \dot{\cup} S_2) \setminus \{r\}$
- $i = \begin{cases} i_1 & \text{if } i_1 \neq r \\ i_2 & \text{otherwise} \end{cases}$
- $A = A_1 \cup A_2$
- $\Omega = ( \quad W_1 \dot{\cup} W_2,$   
 $\quad \text{watch}(z) = \begin{cases} \text{watch}_1(z) & \text{if } z \in S_1 \\ \text{watch}_1(r) \cup \text{watch}_2(z) & \text{if } z \in S_2 \end{cases}$   
 $\quad \text{value}(w) = \begin{cases} \text{value}_1(w) & \text{if } w \in W_1 \\ \text{value}_2(w) & \text{if } w \in W_2 \end{cases} \quad )$
- $T$  is the smallest set defined by:

$$\begin{array}{l}
1 \frac{r \xrightarrow{a, R_1}_{T_1} r \quad t \in S_2}{t \xrightarrow{a, R_1 \cup W_2}_T i_2} \qquad 2 \frac{r \xrightarrow{a, R_1}_{T_1} s \quad s \neq r \quad t \in S_2}{t \xrightarrow{a, R_1 \cup W_2}_T s} \\
3 \frac{s \xrightarrow{a, R_1}_{T_1} r \quad s \neq r}{s \xrightarrow{a, R_1}_T i_2} \qquad 4 \frac{s \xrightarrow{a, R_1}_{T_1} s' \quad s \neq r \quad s' \neq r}{s \xrightarrow{a, R_1}_T s'} \\
5 \frac{t \xrightarrow{a, R_2}_{T_2} t' \quad r \xrightarrow{a, \cdot}_{T_1} \cdot}{t \xrightarrow{a, R_2}_T t'}
\end{array}$$

The definition of refinement replaces the refined state by the refining automaton, taking care to replicate all transitions from the former onto each state of the latter. The negative premise in rule 5 gives priority to transitions exiting the parent state. The initial state of the refining automaton serves as the destination for any incoming, higher-level transitions. Care is taken to reset all timers of the refining automaton on transitions from the parent state, including self-loops. Any timer on the parent state is active whilst making transitions within the refining states. Multi-level transitions are not permitted (their introduction is intricate).

## 2 Interpretation of protocol state machines

Protocol state machines represent interactions across a driver interface. The set of all actions,  $\mathbb{A}$ , is the union of driver dispatch functions and system callbacks (some augmented with return values) in the API. The dispatch functions are

those that end in a question mark, such as `start?` and `txContinue?`, they are triggers from the operating environment where control is passed into a driver. The callbacks are those ending in an exclamation mark, such as `startFailed!` and `txPacketPull[false]!`. They are calls made from the driver to its environment. Since drivers cannot be re-triggered until a dispatch is complete, a function call event and corresponding return event appear as a single action (they occur synchronously from the perspective of a driver).

The semantic mapping of the previous section allows a protocol state machine to be considered as a watched automaton, which in turn is easily reduced to a basic automaton. Each such automaton defines a set of sequences, or valid traces, over the alphabet  $\mathbb{A}$ . A sequence of actions belongs to the set of valid traces iff it is accepted by the automaton.

A protocol state machine may be monitored at run-time by capturing exchanges across the driver interface and tracking the state of the underlying basic automaton. The occurrence of a trigger in a state with no corresponding transition represents an assumption violated by the environment. The occurrence of a call-back in a state with no matching out-going transition represents a fault in the driver. Other driver faults may be detected by monitoring the timers of the watched automaton. A timer  $w$  starts when it is introduced upon entering a state—i.e.  $w \in \text{watch}(i)$  for the initial state, or  $w \in \text{watch}(s') \setminus \text{watch}(s)$  when transitioning between  $s$  and  $s'$ —where it will have the value  $w$ . Timers are made inactive in a similar way, or reset to their initial value when listed against a transition. Timer expiry indicates a driver fault.

## References

- [1] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [2] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1–3):61–92, 2001.