

THE TIMELINE EDITOR 13

*A design without requirements cannot be incorrect.
It can only be surprising.
(Willem L. van der Poel, 1926–)*

Although SPIN provides direct support for the formalization of correctness requirements in terms of linear temporal logic formulae, the use of this logic is not always as intuitive as one would like. The precise meaning of a temporal logic formula is sometimes counterintuitive, and can confound even the experts.

An alternative method, that we will explore in this chapter, is to express properties visually, with the help of a graphical tool. The tool we discuss here is called the *timeline editor*, created by Margaret Smith at Bell Labs. The inspiration for this tool came directly from lengthy discussions on the semantics of temporal logic, which led us to draw many small pictures of timelines on the whiteboard to illustrate sample execution sequences that were either intended to satisfy or to violate a given property. The timeline pictures were so useful that we decided to provide direct tool support for them. The tool was originally envisioned to generate only linear temporal logic formula as its output, but we later found it more effective to generate *never* claim automata in PROMELA syntax that can be used directly by SPIN in verifications.

Technically, the types of properties that can be expressed with the timeline editor tool do not cover everything that can be verified by SPIN, that is, they cover only a small subset of the set of all ω -regular properties. The tool is not even expressive enough to let us specify everything that can be expressed with linear temporal logic, which itself also covers only a subset of the ω -regular properties. Yet, the types of properties that can be expressed seems rich enough to specify many of the types of properties that one needs in system verification in practice. Users of model checking tools often tend to shy away

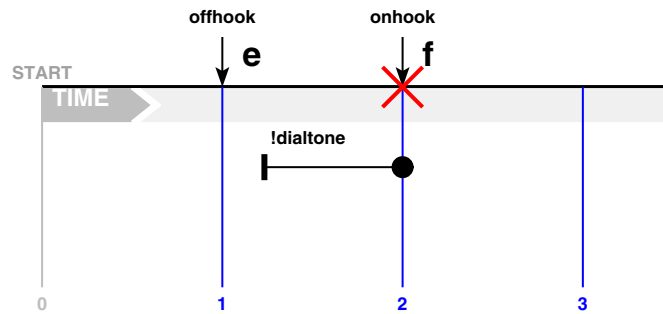


Figure 13.1 Simple Example of a Timeline Specification

from the use of truly complex temporal properties and restrict themselves wisely to a smaller subset of formulae for which it is easier to develop an accurate intuition. The timeline tool appears to capture just that subset and not much more.

The timeline tool allows us to define a causal relation on the events that can occur in a distributed system. It also allows us to restrict the set of sequences that contain the specified events to smaller sets that satisfy additional constraints on specific, user-defined intervals on the timeline. That is, the timeline allows us to select the set of execution sequences that is of interest to us, and then define some correctness criteria for them. The correctness criteria are expressed in the form of events that either must or may not be present at specific points in the execution.

AN EXAMPLE

A first example of a timeline specification is shown in Figure 13.1. It defines two events and one constraint on a system execution. At the top of the drawing canvas is a grey horizontal bar that represents the timeline. Time progresses from left to right along the bar. At regular intervals, there are vertical blue lines, called marks, that intersect the timeline. The first mark, numbered 0, is colored grey and for reference only. The remaining marks indicate points on the timeline where events and constraints can be attached. Marks do *not* represent clock ticks, but are simply used to indicate points of interest during a possibly long system execution. In between two marks any number of execution steps could pass.

Events are attached directly to marks, and placed on the timeline itself. In Figure 13.1 there are two events: *offhook* and *onhook*. Constraints are placed

underneath the timeline, spanning intervals between marks. One constraint, named *!dialtone*, is also shown. During verification the model checker attempts to match each system execution to the events that are placed on the timeline, provided that all corresponding constraints are satisfied. In Figure 13.1, no constraint applies to the occurrence of the first event, *offhook*, but as soon as it has occurred (immediately in the next state), the constraint *!dialtone* must be satisfied for the execution sequence to continue to match the timeline. If eventually, with the constraint still satisfied, the event *onhook* is seen, the timeline is completely matched. Reaching the end of a timeline by itself does not constitute an error condition. In this case, though, an error can be reported because the final event matched on the timeline is a *fail* event.

The requirement specified with the timeline in Figure 13.1 states that it is an error if an *offhook* event can be followed by an *onhook* event without a *dialtone* event occurring first.

For the purposes of property specification, the term *event* is somewhat of a misnomer. Both events and constraints are really conditions (state properties) that must be satisfied (i.e., that must hold) at specific points in an execution. In a PROMELA model, a state property is simply a boolean condition on global state variables that is said to be satisfied when it evaluates to *true*. This means that within the context of SPIN, an event occurrence is not necessarily an instantaneous phenomenon, but can persist for any amount of time. As a simple, though contrived, example, we could define the meaning of event *offhook* in Figure 13.1 to be *true*. This would mean that the event can be detected in any system state, and the event occurrence, as it were, persists forever. If the event persists forever, this merely means that it can be matched at any time during a system execution, so wherever we would place such an event on the timeline, it could always be matched. An event defined as *false*, on the other hand, could never be matched. If we define *onhook* as *false* in Figure 13.1, for instance, then the timeline specification could never be violated, not even if we also define the constraint *!dialtone* as *true*.

TYPES OF EVENTS

There are three different types of events that can be placed on a timeline.

- Regular events are labeled with the letter **e**. If a regular event occurs at a point in a system execution where its occurrence is specified, the execution matches the timeline. If it does not occur, the execution does not match. This does not mean that the execution is in error; it only means that the timeline property does not apply to the non-matching execution.
- Required events are labeled with the letter **r**. A required event can be matched in a system execution just like a regular event. This time, though, it is considered an error if the required event does not appear in the system execution at the point where it is specified, assuming of

course that all earlier events on the timeline were matched, and all applicable constraints are satisfied.

- Failure events are labeled with the letter **f**. Failure events record conditions that should never be true at the point in a system execution where they are specified. It is considered to be an error if a failure event is matched. It is not an error if a failure event does not occur (i.e., is skipped).

Constraints are specified underneath a timeline. Each constraint persists over specific intervals of the timeline. Constraints are denoted by horizontal lines below the main timeline. The start and the end point of each constraint is always associated with a specific timeline mark. Optionally, the constraint can include or exclude the events that are attached to the begin and end marks on the timeline.

There can be any number of events and any number of constraints in a timeline specification, but only one event can be attached to any single timeline mark.

DEFINING EVENTS

Events and constraints are represented by user-defined names on the timeline. The name can contain special characters, such as the negation symbols that we used in the name of the constraint in Figure 13.1. The names can be used directly to generate PROMELA *never* claims, but more typically one will want to define them more precisely to reflect the exact, perhaps more complex, conditions that must be satisfied for the corresponding event or constraint to apply. For the example in Figure 13.1, we can provide definitions for the events *offhook*, *onhook*, and *!dialtone*. The details of these definitions depend on the specifics of the verification model that is used to verify the timeline property. The timeline properties themselves are intended to be definable in a format that is largely model independent. For the final version of the model of a phone system that we develop in Chapter 14, the definitions of the events and the constraint used in Figure 13.1 could be as follows:

```
#define offhook          (last_sent == offhook)
#define onhook          (last_sent == onhook)
#define !dialtone       !(session_ss7@Dial)
```

where the dialtone constraint is specified with the help of a remote reference to the process of type `session_ss7`. There is no real difference in the way that events or constraints are defined. Both events and constraints define state properties: boolean conditions on the system state that can be evaluated to *true* or *false* in any reachable system state of the model. Only their relative placement on a timeline determine their precise semantics, that is, whether they are used to act as events to guide the matching of system executions, or as constraints to restrict the types of executions that can match.

MATCHING A TIMELINE

The verification of a timeline proceeds as follows. In the initial system state, the first mark on the timeline is designated as the *current mark*. At each execution step of the system, the verifier evaluates the event condition attached to the current mark on the timeline, and it evaluates all constraint conditions attached to intervals that intersect the blue vertical line for this mark. If the next event to be matched is a failure event, then the event that follows it on the timeline, if any, will also be evaluated. If a condition evaluates to *true*, the corresponding event or constraint is said to be *matched*; otherwise, it is not matched. The context now determines what happens next. There are several possibilities.

- The current execution sequence no longer matches the timeline specification because a constraint condition is now violated. The verification attempt for this sequence can be abandoned.
- If all constraint conditions are satisfied and the event condition at the current mark is matched and that event is a failure event, an error can be reported.

If under the same conditions the event at the current mark is not a failure event, the current mark is advanced to the next mark on the timeline, and the verification is repeated after the next system execution step is performed.

- If all constraint conditions are matched, but the event condition is not matched and the current event is not a failure event, then the current mark remains where it is, and the verification is repeated after the next system execution step is performed.

If under the same conditions the current event is a failure event, and the next event on the timeline, if any, is matched, the current mark moves to the event that follows that next event, and the verification is repeated after the next system execution step is performed. The timeline tool does not permit two adjacent failure events on a timeline, so an event that follows a failure event is either a regular or a required event.

- If the end of an execution sequence is reached, but the end of the timeline has not been reached *and* the event at the current mark of the timeline is a required event, an error can be reported for this execution sequence.

If the end of a timeline is reached before the end of an execution, the verification effort can also be abandoned, since no further errors are possible.

AUTOMATA DEFINITIONS

The Büchi automaton that corresponds to the timeline specification from Figure 13.1 is shown in Figure 13.2. The automaton has three states, one of which, state s_2 , is accepting. The initial state of the automaton is s_0 .

The automaton can be generated automatically by the timeline editor either in graphical form, as shown in Figure 13.2, or in PROMELA syntax as a `never`

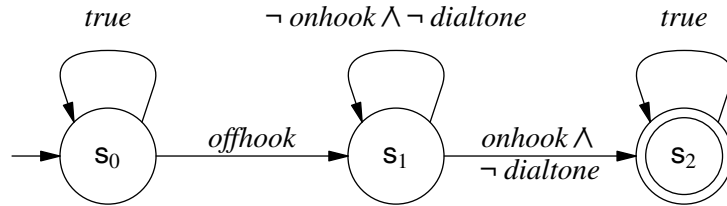


Figure 13.2 Büchi Automaton for the Timeline in Figure 13.1

claim. The PROMELA version of the automaton is shown in Figure 13.3.

The second state of the automaton can only be reached if an *offhook* event occurs, which is followed by an interval in which *dialtone* remains *false* and no *onhook* event is detected. Then the transition to the accepting can be made if an *onhook* event occurs, still in the absence of a *dialtone* event. Once the accepting state is reached, the remainder of the run is automatically accepted due to the self-loop on *true* in state S_2 : the violation has already occurred and can no longer be undone by any future event.

```
#define p1 (last_sent == offhook)      /* offhook */
#define p2 (last_sent == onhook)      /* onhook */
#define p3 !(session_ss7@Dial)        /* !dialtone */

never {
S0:   do
      :: p1 -> goto S1
      :: true
    od;
acceptF0:
      assert(0);
      0;
S1:   do
      :: p2 && p3 -> goto acceptF0
      :: !p2 && p3
    od;
}
```

Figure 13.3 Never Claim for the Timeline in Figure 13.1

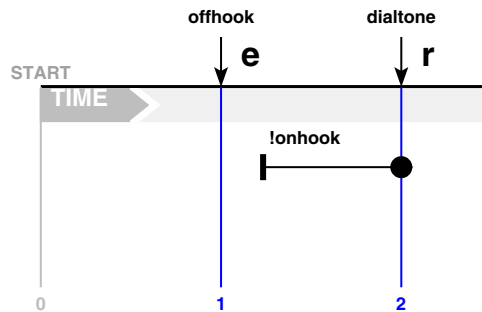


Figure 13.4 Variation on the Timeline from Figure 13.1

CONSTRAINTS

A constraint interval always has one of four possible forms, depending on whether the start and the end points of the interval are included or excluded from the constraint. By adding constraints, we never really modify the structure of the Büchi automaton, or of the PROMELA *never* claim, that corresponds to a timeline. Added constraints can only restrict the number of sequences that can be matched at each step of the timeline, by adding conditionals to the transitions of an existing automaton structure.

VARIATIONS ON A THEME

We have not said much about the rationale for the property that is expressed by the timeline specification from Figure 13.1. Informally, the property states that it would be an error if there can exist execution sequences in which an *offhook* event can be followed by an *onhook* event, without *dialtone* being generated in the interim. It may of course be possible for a telephone subscriber to generate a fast *offhook*–*onhook* sequence, but we may want to use the timeline specification to inspect precisely what happens under these circumstances by generating the matching execution scenarios.

We can also attempt to express this property in a different way. There can be small differences in semantics, depending on whether conditions are used as events or as constraints. As a small example, consider the variant of this property that is shown in Figure 13.4. We have switched the roles of *dialtone* and *onhook* as event and constraint here, compared to Figure 13.1.

At first blush, this timeline appears to express the same property, this time labeling the appearance of *dialtone* after an *offhook* as a required event, and the absence of *onhook* as a constraint.

We can see more clearly what is required to match this timeline by inspecting

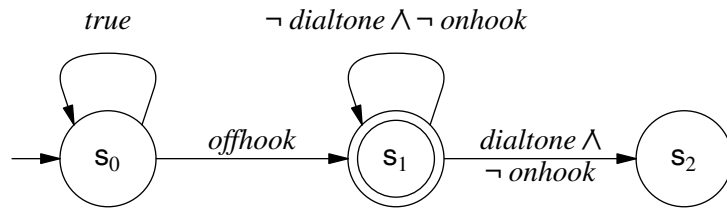


Figure 13.5 Büchi Automaton for the Timeline in Figure 13.4

the corresponding automaton structure, as shown in Figure 13.5.

This time, state s_1 is the Büchi accepting state. The only way for an execution sequence to trigger an error would be if it contained an *offhook* event that is never followed by either an *onhook* or a *dialtone* event. When state s_2 is reached instead, the requirement expressed by the timeline specification is satisfied, and no further errors can result. This means that, technically, state s_2 , and the transition that leads to it, is redundant and could be omitted from the automaton without changing its meaning.

Assume now that there were an execution of the switch system we intended to verify that would occasionally fail to give *dialtone* after an *offhook*. Very likely, both in a verification model and in real life, the unlucky subscriber who encounters this behavior will not remain *offhook* forever, but eventually return the phone *onhook*. This means that the error, if present, would not be caught by this specific variant of the specification, unless we explicitly model behavior where the subscriber can permanently keep the phone off-hook.

In reality, the *dialtone* property for a telephone switch has both a functional and a real-time performance requirement. *Dialtone* should not only be generated after an *offhook* event, but on average also follow that event in 0.6 seconds. In 98.5% of the cases, further, *dialtone* should appear within 3 seconds after an *offhook*. Since timelines and SPIN models target the verification of only functional system requirements, the real-time performance aspects of requirements cannot be captured or checked in this way.

TIMELINES WITH ONE EVENT

There are only two useful types of timelines that contain one single event. A timeline with a single regular event is not of use, since it does not express any requirement on an execution. That is, the timeline might match an execution that contains the event that is specified, but no matching execution can ever be flagged as erroneous in this way. The two smallest timelines of interest are the ones that contain either a single required or a single fail event, as

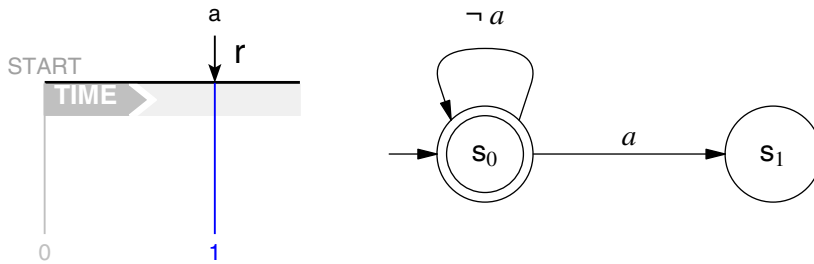


Figure 13.6 Timeline and Automaton for a Single Required Event

illustrated in Figures 13.6 and 13.7.

The timeline specification from Figure 13.6 traps a system execution error in the same cases as the LTL formula that we would use to express the violation of a system invariant property

$$\Box \neg a = \neg \Diamond a.$$

The timeline specification in Figure 13.7, similarly, traps a system execution error in the same cases as the property

$$\Diamond a.$$

These first two properties can be seen as duals: one requires the absence of an event, and the other requires at least one occurrence.

TIMELINES WITH MULTIPLE EVENTS

With two events, we can form five different types of timelines. Each of the two events can be one of three different types, but clearly four of the nine possible combinations are not meaningful. A timeline with two regular events, for instance, cannot fail any system execution to which it is applied. Further, if the last event of the timeline is a regular event, then that event would always be redundant. And, finally, a timeline with two fail events that are placed on adjacent marks has no reasonable semantics, and is therefore rejected by the timeline tool. (In this case the conditions for the two fail events should probably be combined into a single condition.)

One of the five remaining meaningful combinations of two events is reproduced, with the corresponding automaton, in Figure 13.8.

The timeline property from Figure 13.8 is similar, though not identical, to the LTL response property:

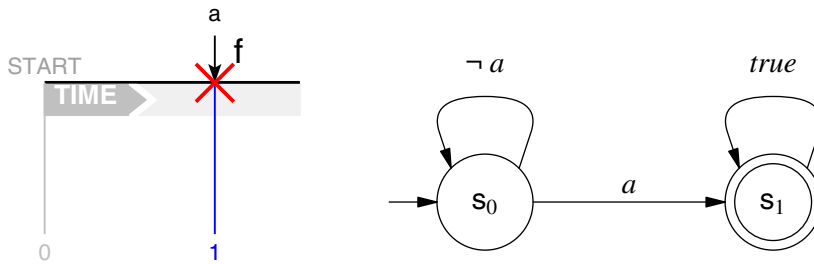


Figure 13.7 Timeline and Automaton for a Single Fail Event

$$\neg(a \rightarrow \diamond b).$$

Note that the LTL property requires condition a to hold at the start of each sequence, since it is not preceded by a temporal operator. The timeline specification does not have that requirement. The LTL formula that precisely captures the timeline property from Figure 13.8 is somewhat more complex, namely:

$$\neg(\Box(a \rightarrow X(\diamond b))).$$

The example timeline in Figure 13.9 contains three of the five possible combinations of two events.

We have labeled the four events on this timeline with letters from a to d , and added a constraint named z . A specification of this type could be used to check one of the requirements for the implementation of call waiting on telephone lines. Event a could then represent the occurrence of an incoming call on a line that is currently involved in a stable two-party call. The requirements state that with the call waiting feature in effect, the subscriber should at this point receive a call waiting alert tone, which would correspond to event b . Provided that none of the parties involved abandon their call attempts, or take any other action (which can be captured in constraint z), the first alert tone must be followed by a second such tone, but there may not be more than these two alerts. So, events b , c , and d would in this application of the timeline all represent the appearance of a call waiting alert tone, which is required twice, but erroneous if issued three or more times. The Büchi automaton that corresponds to the timeline from Figure 13.9 is shown in Figure 13.10. There are three accepting states corresponding to the three different ways in which this timeline specification could be violated: either one of the two required events

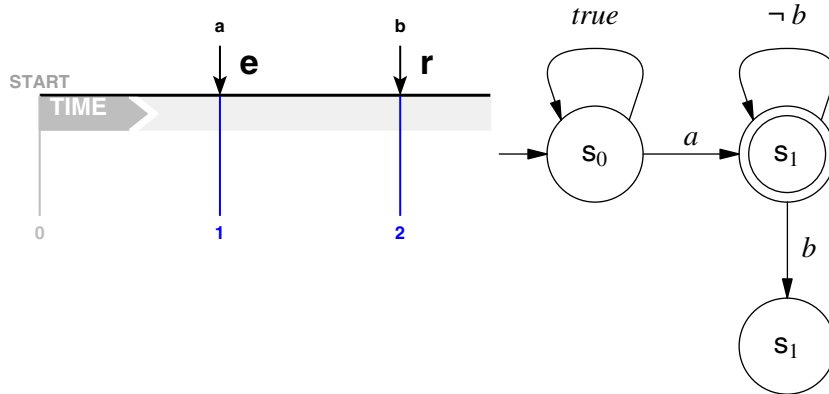


Figure 13.8 Timeline and Automaton for a Regular and a Required Event

could fail to show up, or the failure event could appear when it should not.

Another type of timeline, with the remaining two possible event combinations, is shown in Figure 13.11. This time, a failure event precedes a required event, indicating that after the optional occurrence of the event named *a*, the occurrence of *c* is required, and the occurrence of *b* forbidden.

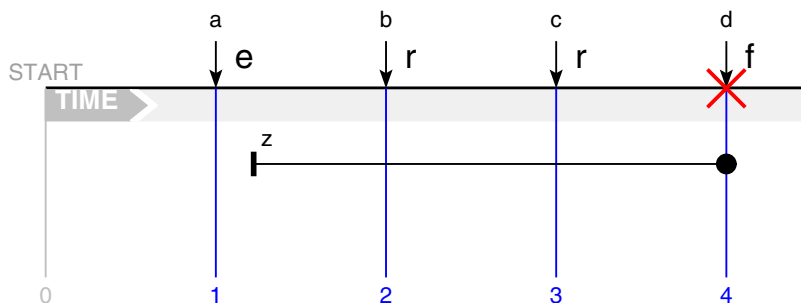


Figure 13.9 A More Complex Timeline Specification

A timeline specification of this type may be used to check the property that *offhook* and *onhook* events must always alternate. We can achieve this by

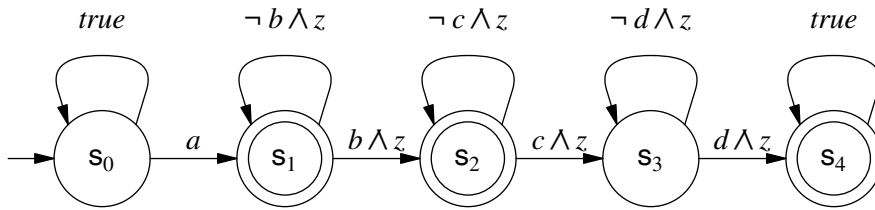


Figure 13.10 Büchi Automaton for the Timeline in Figure 13.9

defining events a and b both as *offhook* events, and event c as an *onhook* event. Constraint z then can restrict the executions that are considered for compliance with this requirement to those where no *onhook* event appears in the interval between a and b .

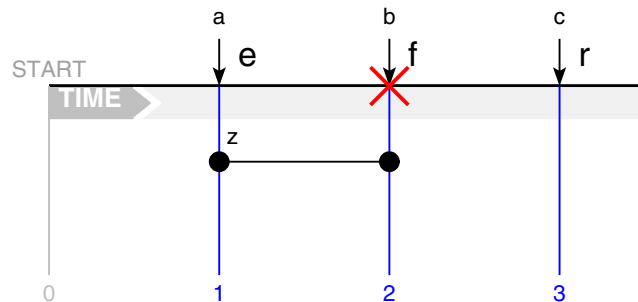


Figure 13.11 Timeline Specification with Three Events

The Büchi automaton corresponding to the timeline from Figure 13.11 is shown in Figure 13.12. The automaton has four states, two of which are accepting. Note again that the automaton is not necessarily completely specified. There is, for instance, no transition out of state s_1 if simultaneously $(\neg z \wedge \neg c)$. In this case, we have just passed the first step of the timeline, and wait for either c or b to occur while z remains *true*. If neither event c nor event b occurs in a given run and constraint z is no longer satisfied, the automaton can stop tracking the run, since no violation matching the timeline is possible in the remainder of this execution.

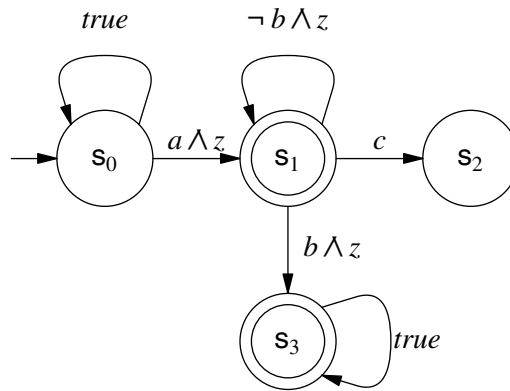


Figure 13.12 Büchi Automaton for the Timeline in Figure 13.11

THE LINK WITH LTL

It is not hard to show that for every timeline specification there exists a formalization in LTL, but the reverse is not necessarily true. Timelines are strictly less expressive than linear temporal logic, and therefore they are also less expressive than ω -automata (which includes PROMELA *never* claims).

Consider, for instance, the LTL formula: $\!(a \text{ U } b)$. The positive version of this requirement would match any run in which a remains *true* at least until the first moment that b becomes *true*. If b is already *true* in the initial state, the requirement is immediately satisfied. The negation of the requirement matches any run where the positive version is violated. This means that in such a run b cannot be *true* in the initial state, and a must become *false* before b becomes *true*.

This seems like a requirement that we should be able to express in a timeline specification. The timeline we may draw to capture it is shown, together with the corresponding Büchi automaton, in Figure 13.13. We are slightly pushing the paradigm of timeline events here, by putting a negation sign before the name of a timeline event. Doing so, we exploit the fact that at least in the context of SPIN an event is really a state property that can be evaluated to *true* or *false* in every reachable system state.

Unfortunately, the automaton that is generated does not precisely capture the LTL semantics. The correct Büchi automaton, generated with SPIN's built-in LTL converter, is shown in Figure 13.14. It closely resembles the timeline automaton from Figure 13.13, but it is not identical. In the correct version, the self-loop on state s_0 requires only b to be false, but makes no requirement on the value of a .

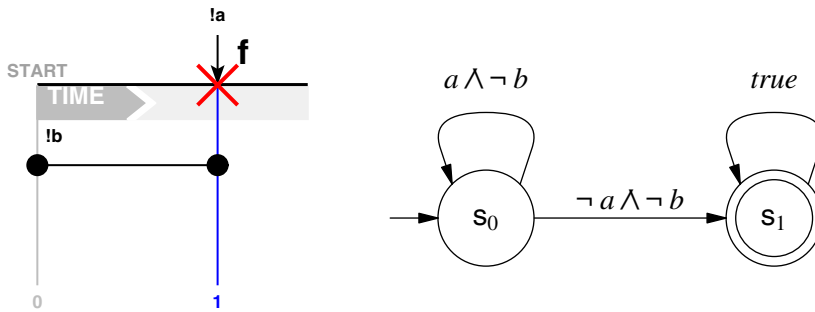


Figure 13.13 Attempt to Express the LTL property $!(a U b)$

In the timeline from Figure 13.13 we used a negation sign in front of an event symbol, in an attempt to capture the semantics of the LTL until property. If we go a little further and use arbitrary boolean expressions as place holders for events, we can create many more types of timelines. As just one example, consider the timeline that is shown in Figure 13.15. Although it looks very different from the timeline from Figure 13.13, it turns out to define precisely the same property.

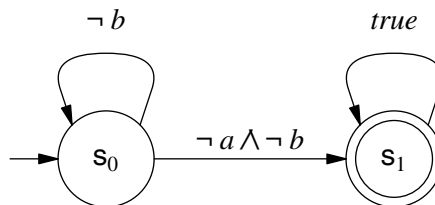


Figure 13.14 The Correct Büchi Automaton for LTL property $!(a U b)$

Fortunately it is not hard to check and compare the semantics of timeline descriptions by using the timeline editing tool to generate the corresponding Büchi automata. The automaton that corresponds to the timeline from Figure 13.15, for instance, is identical to the one shown in Figure 13.13. It can be very hard, though, to reason backwards, and to find the proper timeline specification for a given Büchi automaton, such as the one from Figure 13.14, assuming, of course, that one exists.

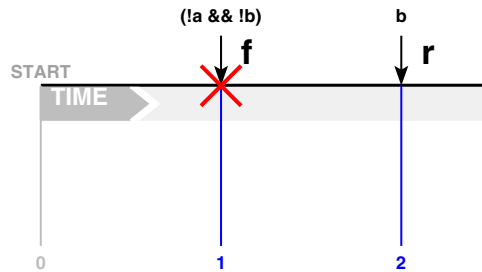


Figure 13.15 A Variant of the Timeline in Figure 13.13

Timelines can easily be used to express system safety properties, but they can only express a small class of liveness properties. The only mechanism from timeline specification that we can use to express liveness properties is the notation for a required event. The relatively simple visual formalism from a timeline specification, though, appears to suffice for handling most cases of practical interest in systems verification.

An often heard criticism of LTL is, for instance, that the true meaning of formulae with more than two or three temporal operators can be very difficult to understand, even by experts. Similarly, accurately capturing the correct semantics of a complex temporal property in an LTL formula can be a daunting task. Some interpret this to mean that we should shy away from the complex formulae in systems verification. In this respect, the lack of expressiveness of the timeline editing tool may well be regarded a strength.

“Mathematicians are like Frenchmen: whenever you say something to them they translate it into their own language, and at once it is something entirely different.”
J.W. von Goethe (1749–1832)

BIBLIOGRAPHIC NOTES

Several other visual formalisms for specifying systems and properties have been proposed over the years. The best known such proposals include Harel [1987] for systems specifications, and Schlor and Damm [1993] or Dillon, Kutty, Moser, et al. [1994] for property specification.

An alternative method to make it easier to capture complex logic properties as formulae in temporal logic is pursued by Matt Dwyer and colleagues at Kansas State University. Dwyer, Avrunin, and Corbett [1999] describe the

design and construction of a comprehensive patterns database with formula templates for the most commonly occurring types of correctness properties.

Information for downloading the timeline editor, which is freely available from Bell Labs, can be found in Appendix D.