Events and Constraints:
A Graphical Editor for Capturing Logic Requirements of Programs

Margaret H. Smith
Bell Laboratories
Rm. 2C-407
600 Mountain Avenue
Murray Hill, NJ 07974
mhs@research.bell-labs.com

Gerard J. Holzmann
Bell Laboratories
Rm. 2C-522
600 Mountain Avenue
Murray Hill, NJ 07974
gerard@research.bell-labs.com

Kousha Etessami
Bell Laboratories
Rm. 2C-472
600 Mountain Avenue
Murray Hill, NJ 07974
kousha@research.bell-labs.com

Abstract

A logic model checker can be an effective tool for debugging software applications. A stumbling block can be that model checking tools expect the user to supply a formal statement of the correctness requirements to be checked in temporal logic. Expressing non-trivial requirements in logic, however, can be challenging.

To address this problem, we developed a graphical tool, the TimeLine Editor, that simplifies the formalization of certain kinds of requirements. A series of events and required system responses are placed on a timeline. The user converts the timeline specification automatically into a test automaton, that can be used directly by a logic model checker, or for traditional test-sequence generation.

We have used the TimeLine Editor to verify the call processing code for Lucent's PathStar Access Server against the TelCordia LSSGR standards. The TimeLine editor simplified the task of converting a large body of English prose requirements into formal, yet readable, logic requirements.

Keywords
model checking, software verification, testing, requirements.

1. Introduction

Logic model checkers are gaining in popularity as tools for debugging concurrent, distributed software. While they cannot completely replace traditional testing tools, for certain types of software bugs, namely control logic and communications errors, model checkers are unequalled in their speed and coverage [5].

The basic steps involved in the application of a model checker to a software application, shown in Figure 1 are:

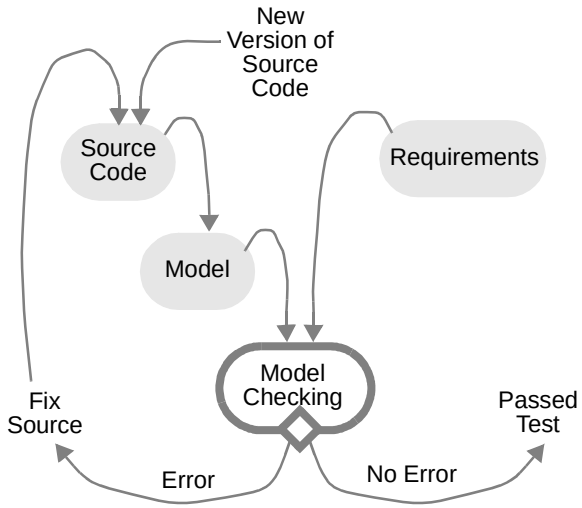


Figure 1. Model checking process

1. obtain a model of the source code.
2. obtain the requirements to be checked.
3. perform model checking step (automated).
4. evaluate any error traces that the model checker generates, determine if the error is in the code or the requirements, and repeat the process if needed.

Defining the requirements to be checked (step 2.) can be an optional step. In the absence of specific requirements to be checked, a model checker will check for some generally desirable program requirements such as: absence of deadlock, livelock, and unreachable code. However, if one wants to check specific requirements, or in other words, check that the source code satisfies application-specific requirements, then some requirements must be defined.

The telephone system is by its very nature a distributed application, and therefore telephony software is a natural candidate for model checking applications. These systems typically support hundreds of distinct and potentially interacting features, which makes their complexity far exceed human reasoning abilities.

If we want to determine, for instance, that a telephone switch satisfies the simple requirement:

when user goes offhook the system provides dialtone

a model checker can exhaustively test whether there is any possible interleaving of events that can lead to an error (i.e., a violation of the stated requirement).

We might attempt to test the same requirement using traditional testing. To do so, we could try to alter the state of the switch in as many ways as possible and go offhook (i.e., pick up the handset) to determine whether we get dialtone in every case. At best such a method samples the possible behavior of the switch. A lucky tester will discover an exception to the requirement, but many bugs pass through, to be discovered in the field by users.

The benefits of model checking for exhaustively testing distributed code should be clear, however, one of the hurdles that practitioners of model checking face is that the requirements must be stated strictly formally. A necessary step is to restate informal requirements, such as our example requirement regarding dialtone, in a formal notation. When the requirement is stated formally, we call it a *requirement*.

The formal notation of choice for specifying requirements of software applications is Linear Temporal Logic (LTL) [12]. LTL allows one to describe how a system's events and states are related over time, which is exactly what we need for expressing logic control and communications requirements. For example, we can use an LTL formula to formally state the simple requirement above, but also more complex requirements about required causality in the system.

A strength of LTL is that it is expressive, allowing the formalization of vastly more complex requirements than the one we expressed. A drawback of LTL is that it is hard to debug an LTL formula, even for experts. In an industrial application of model checking to the Lucent PathStar Access Server, we were faced with the challenge of specifying and checking the code for conformance with 117 distinct feature requirements. We had two alternatives for overcoming this hurdle. The first was to capture all requirements in logic, and to reserve adequate time for debugging the LTL formulas. The second was to find a more natural way to generate the required formal requirements without writing LTL formulas.

To support the second approach, we built a graphical TimeLine Editor that can generate formal requirements from a visual representation of required causal relations. While not as expressive as LTL, the TimeLine Editor allowed us to specify a large fraction of the feature requirements of interest in the verification of the PathStar Access Server [9]. The time spent building the editor was well rewarded by the time savings realized.

The TimeLine Editor is well suited for expressing the types of requirements encountered in the PathStar application. Specifically the TimeLine Editor can express requirements with a preamble (a sequence of events that act as a pattern to be matched against execution sequences) and a response. Other visual notations [1][13] have been developed to address the expressive needs of different applications, or to express a broader set of requirements. Other related work [7] has been to classify and codify certain frequently observed requirement types.

To explore the TimeLine Editor further we will look at how we can discover requirements, typical forms of requirements, the timeline notation and graphical interface, how to convert a timeline to a test automaton, the types of requirements that can and can not be expressed using timelines, global requirement constraints, and an example error that was found using the TimeLine Editor and the Feaver/Spin model checking framework.

2. Discovering requirements

Many efforts have been made to apply various requirements modeling tools in systems engineering. Despite this, the practical reality is that most requirements are today still expressed in English prose. Such was the case for the Bellcore (now TelCordia) LSSGR standards [9] that served as requirements for the PathStar Access Server. In these documents, the requirements are not enumerated, or marked, but described in a white paper format.

We thoroughly explored the standard for each feature to find requirements that were amenable to model checking. Such requirements must be testable and they must describe the required temporal behavior of the system. To be testable, a requirement must describe some aspect of the system that is observable and the outcome of the requirement must be measurable. Not every requirement will meet this criterion. For instance, in the LSSGR standard for Call Waiting (CW), this requirement appears:

> *"The number of special circuits (if any) that are used by a switching system in providing CW should be an engineered quantity."*

How do we test this requirement? From this requirement we learn that a switching system may or may not have 'special circuits'. We can guess that traffic measurement and application of formulas are involved in determining exactly how many special circuits, if any, are needed, but this requirement doesn't give us data or the necessary formulas. Therefore, we conclude that this requirement

cannot be tested with model checking because there is no measurable outcome.

To be amenable to model checking, a requirement must also specify the temporal behavior of a system; that is, how the system acts in response to external stimuli and internal conditions over time. Even if the 'special circuit' requirement had described an observable and measurable aspect of the system, it clearly does not pass the second criteria because it does not describe temporal behavior.

Some of the features we tested were two user features and many were three or more user features. The number of users was the greatest determinant in the complexity of the feature because as the number of users increased the possible combinations of user events increased exponentially. For instance, at any given point in feature processing, any party on the call can go onhook, and for the requirements to be complete, there must be a specific system response requirement for each user's onhook event at each state of feature processing.

For the less complex features, such as *Call Forwarding* (forwarding an incoming call to an alternate destination under certain circumstances -- i.e. no answer, busy, etc.) or *Denied Originating Service* (denying the subscriber the ability to initiate calls), it was straightforward to understand the temporal behavior and the requirements could be written directly. For features with more than two users: *Call Waiting, N-Way Calling, Hold, Transfer*, in order to understand the behavior of the feature it was necessary to build an informal model of the intended feature behavior, e.g., in a graphical editor. An informal model, such as the one for a portion of the *Call Hold* feature shown in Figure 2, can be used to make sure that one has captured all the possible events that can occur and is useful for identifying relevant constraints on requirements (as will be described in Section 4.).

When the informal model is complete, we can select paths through the graph that reflect the critical aspects of the intended behavior. Each path becomes a requirement that can be checked in detail against the source code. A path in the informal model is a sequence of events, where an event is either generated by the system or by a test harness. In Figure 2, the system events are shown on a gray background. The cp refers to the controlling party, and the hp refers to the held party.

In the verification of the PathStar Access Server we devised a mechanism that allowed us to automatically generate a formal model directly from the source code of the application (which was written in C) [6]. The model extraction tool converts C source code into the input language of an efficient software model checking tool, Spin [5]. Spin uses the formal requirements (expressed in temporal logic or with the help of the TimeLine Editor tool) to fine-tune the program model with a slicing technique. The slicing algorithm in Spin uses data dependency analysis and control flow analysis to

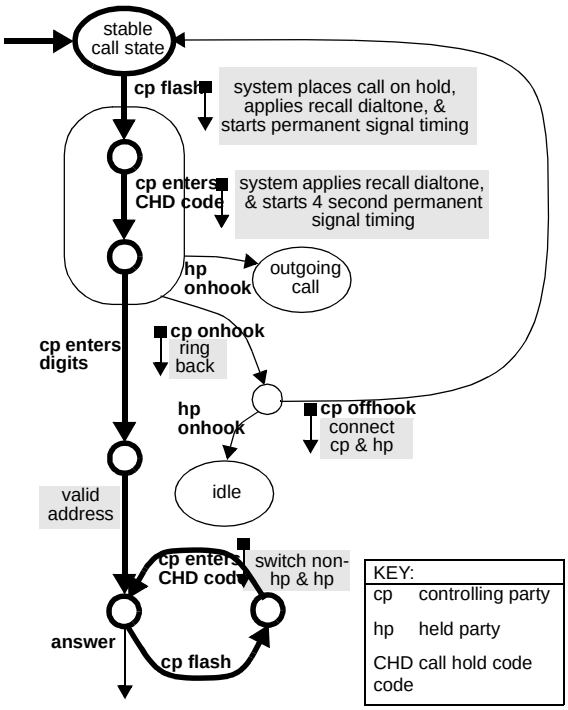


Figure 2. A partial model of the Call Hold feature, with a requirement, depicted as a path

automatically abstract away details of the model that are not relevant to the requirement being checked, while retaining the parts of the model that are relevant to the requirement. The abstraction that is employed here has the important property that it is logically *conservative*, which means that if the abstracted model can be shown to satisfy the correctness requirement, then the original source program necessarily also satisfies that requirement.

Using the requirements to guide the generation of the abstract model of the source code helps to ensure that the levels of abstractions in the requirement and the model match. For instance, if the requirements are concerned with the fact that a full digit string has been entered rather than the particular digits in the digit string, then in the model extraction step we can automatically abstract away most of the details of the digit analysis code so that we only retain the possible outcomes: invalid digits, partial digits or full digit string.

In total we analyzed 17 Telcordia LSSGR feature standards. On average we identified seven requirements to test per feature. The feature with the largest number of test requirements identified was Call Waiting, with 13 test requirements. The large number of testable requirements for Call Waiting reflects the complexity of the feature, and the thorough and complete requirements for Call Waiting, as compared to the other features we analyzed.

3. Typical forms of requirements

As in traditional testing, when applying model checking of distributed software, it is necessary to configure the system and the test driver elements before a test can be performed. In traditional testing one would drive the system into a particular state of interest by feeding it a sequence of events, called the *preamble*. After the preamble, the tester awaits the expected response. In model checking the preamble becomes a pattern that is matched against the executions of the system.

In model checking it is also possible to make use of known, well defined system states in order to reduce the number of events in the preamble. In a telephone switch, for instance there are well-defined so-called *stable call-states* such as busy, idle, dial, and 3-way call. There may be *many* different system executions that lead into one of these states, so using the well-defined state as the first element to be matched in the preamble, as opposed to a specific execution that leads to that state, has the effect of generalizing the requirement, thereby broadening and strengthening the check.

We can see how a system state is used in formulating a requirement by selecting a path through the fragment of the *Call Hold* model. The Call Hold feature allows its subscriber to place an active (non-held) party on hold in order to initiate a call to yet another party. The path we are interested in begins at a stable call state, which is defined as a call consisting of the controlling party (the party who subscribes to the Call Hold feature) and another party, in which no change to the state of the call's connection is imminent. The cp refers to the controlling party, or the party that is using Call Hold to initiate a new call, and the CHD code is the Call Hold Code, an assigned sequence of digits that invokes the Call Hold feature. For the path through the *Call Hold* model, the preamble consists of these events:

cp flash, cp enters CHD code, cp enters digits, cp flash, cp enters CHD code

The initial event of the preamble: cp flash, occurs during the stable call state, and the required system response is that the non-held and held parties should be switched.

The requirement defined by the highlighted path should *pass* the check. This means that if the model checker can find any execution where the initial state and preamble occur, but the required event does not, this will be reported as a violation of the requirement.

Just as in traditional testing, in model checking the requirement is not useful if the preamble is not correct. In model checking an incorrect preamble could give a vacuously positive result because it might not match any execution in the system. We can debug the preamble by asking the model checker to find at least one sequence that matches the preamble. If the model checker finds a match, then we know that our preamble is not vacuous. If the model checker does not find such an execution, the preamble is likely to be incorrect and we can reconsider its definition.

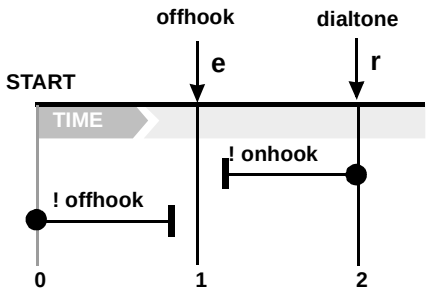


Figure 3. Timeline requirement: 'when the user goes offhook the system should provide dialtone'

Describing an accurate preamble can be cumbersome in temporal logic. In LTL, chains of events are most naturally expressed by a continued functional nesting of *Until* sub formulae. For instance, the simple requirement correlating *offhook* and *dialtone*, without intervening *onhook* events, shown graphically in Figure 3, is stated in LTL as:

!(!offhook U (offhook /\ X [](!dialtone /\ !onhook)))

If we wanted to add additional events between the offhook event and the response, dialtone, each event **i** would require the inclusion of an additional nested Until subformula of the following form:

X((!event $_i$ /\ !onhook) U (event $_i$ /\ !onhook))

The addition of events quickly makes the LTL version of the requirement long and difficult to understand.

The operators used in this LTL formula are as follows. [] : always, X: next, U: strong until, !: logical NOT, /\: logical AND.

4. Description of timeline notation

The timeline notation arose during the early phases of the project when the verification team members were writing and debugging LTL requirements for the verification effort. To clarify what was meant by a certain LTL requirement, a timeline diagram would be drawn on the board. Once it was observed that the timeline diagram could express the requirements of interest to the application, and that it was possible to automatically convert the timeline diagrams directly to Büchi automata, a decision was made to build the TimeLine Editor tool.

A timeline is represented by a wide horizontal bar, as illustrated in Figure 3, with time progressing from left to right. Descending from the timeline bar are vertical bars, called *marks*, which mark the interesting event occurrences, ordered in time. The events can be generated

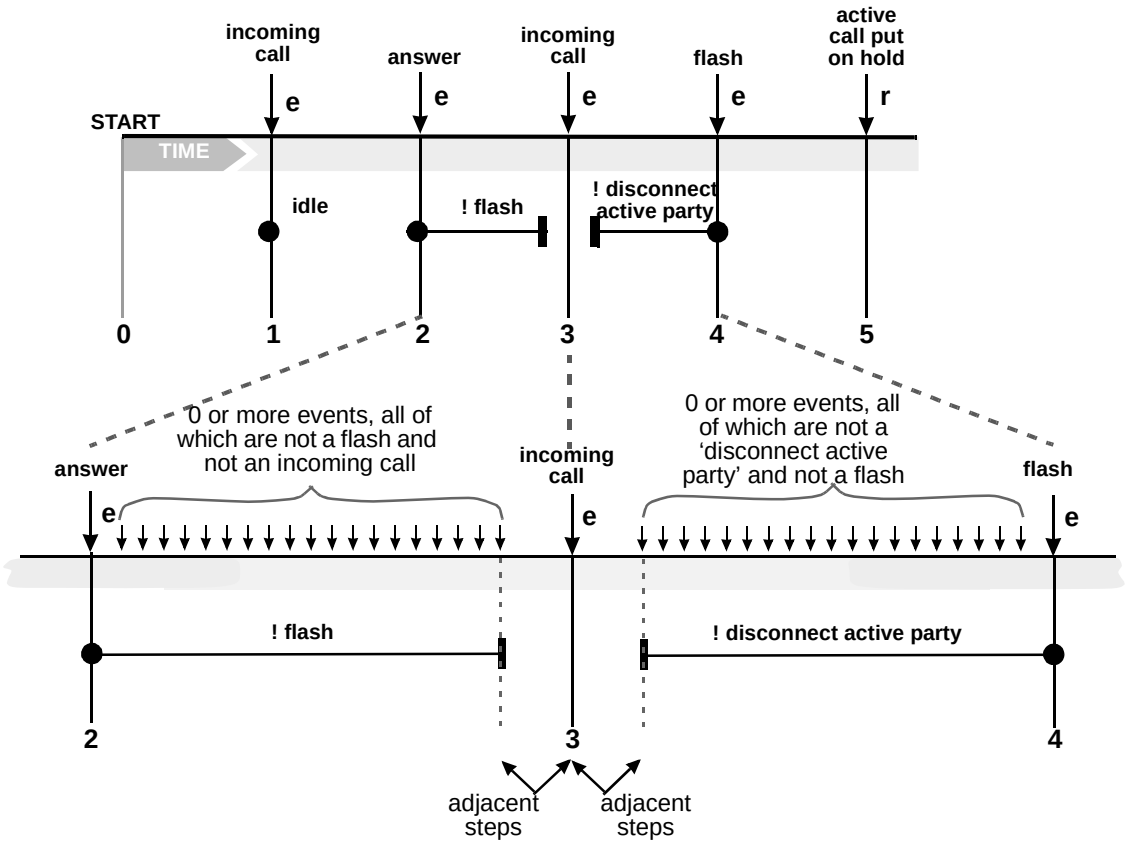


Figure 4. Exclusion of endpoints in a constraint.

anywhere in the system, by any one of many concurrent processes in the distributed system. Therefore, no fixed time-interval can be assumed between subsequent marks (there is no hidden assumption of a "global clock"). There are three types of system events that can be indicated on the timeline.

Regular events - denoted by the letter **e**. These are optional events, that are used to identify the precise executions of the system that we are interested in. For a switch, a regular event could be the user going offhook, flashing the hook, or the arrival of an incoming call. Most regular events are generated by test harness components, that is, the stubs of components external to the system under test.

Required events - denoted by the letter **r**. These are events that are required to occur if all previous events (regular and required) on the timeline have occurred, under the applicable constraints (more about constraints later). It is an error if is possible for a required event to be absent from an execution under these circumstances. For a switch, a required event might be the generation of *dialtone*, or the forwarding of a call.

Fail events - denoted by the letter **f** and a red **X**. These are events that should not occur if all previous events (regular and required) on the timeline have occurred, under the applicable constraints. It is an error if a fail event occurs. For a switch, a fail event might be generation of reorder tone when the user goes offhook

Since it is an error for the system to give reorder tone (fast busy) in response to the user going offhook, we can add reorder as a fail event between the offhook and dialtone events, to strengthen the requirement, as shown in Figure 5. Now this requirement states that after we detect offhook, and while we are waiting for dialtone, if we detect reorder tone, an error has occurred

In addition to events, there are also *constraints,* which are black horizontal lines positioned beneath the timeline bar. We can use constraints to specify that we are not interested in the occurrence of particular events over certain intervals of the requirement. For instance, if there is a requirement that the system must respond to an *offhook* by providing *dialtone*, we can specify the constraint *!onhook* for the interval between the *offhook* and the *dialtone* event.

A constraint begins at one mark and ends at the same or at a subsequent mark. A constraint can include or exclude the marks where it begins or ends. Figure 4 shows a

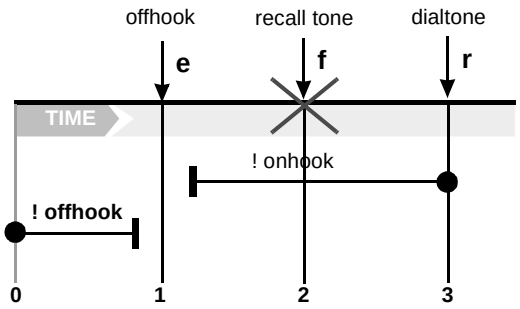
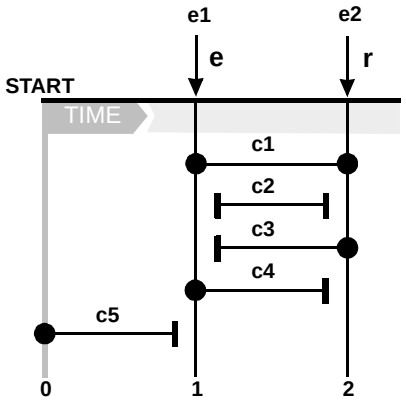


Figure 5. The requirement in Figure 3 is strengthened by the addition of the fail event, "recall tone"'



constraint, **! flash**, that excludes its end mark, and another constraint **! disconnect active call** that excludes its begin mark. If a constraint endpoint includes a mark, this is indicated by a filled circle. If a constraint excludes a mark the end point of the constraint will not overlap the mark but will terminate with a short vertical bar near the mark.

If a constraint includes its begin mark then the constraint applies from the moment that the event attached to this mark occurs. If a constraint excludes its begin mark, then the constraint does not apply when the event attached to this mark occurs (but it still may hold unless expressly stated otherwise by another constraint) but it applies at the event immediately following the event attached to the begin mark. Likewise with a constraint that excludes its end mark; the constraint applies until the event immediately preceding the event attached to the constraint's end mark. If the constraint includes its end mark, then the constraint applies before the event attached to its end mark occurs and when the event attached to its end mark occurs.

A constraint that is indicated by a single filled circle with no horizontal line, such as the **idle** constraint in Figure 4, holds only for the event to which the constraint is attached.

A constraint that begins at the START mark, such as constraint **c5** in Figure 6, applies from the beginning of the execution. Constraints **c1**, **c2**, **c3**, and **c4** of Figure 6 show the constraint variations that may apply between and before two non-fail timeline events **e1** and **e2,** and the corresponding automata. A constraint may not begin or end at a fail event, unless the fail event is the first event or last event of the timeline. A constraint may intersect a fail event as in the case of constraint **! onhook,** that intersects the fail event **recall tone** on Figure 5.

In the requirement shown in Figure 4 for the *Call Waiting* feature, the first four events serve as the *preamble*; the part of the requirement that in effect drives the system into the state of interest. The initial step of interest in the requirement is that the subscriber is in the idle state (a

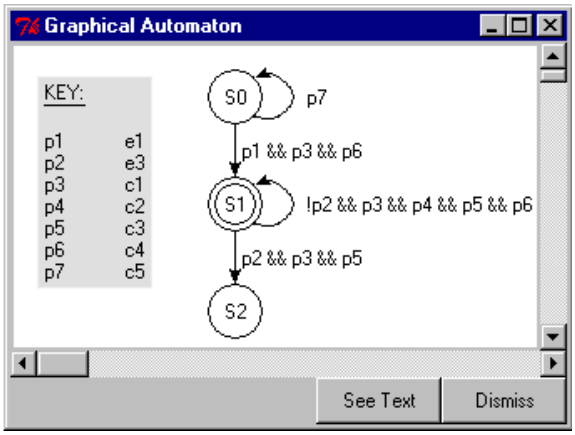


Figure 6. Additional constraint variations

stable call state) when an incoming call arrives. The subscriber then optionally answers the incoming call. If this occurs, and a second incoming call arrives, the subscriber can optionally respond by flashing the hook to invoke the *Call Waiting* feature. The system is then required to respond by putting the active call on hold.

Because we want to check the *Call Waiting* feature in particular, we may want to avoid checking executions that include a flash before the second incoming call arrives. The reason for this is that if the user flashes before the second call arrives, the system can consider the flash to be an invocation of another feature such as *Call Hold* or *Three-Way Calling*. By excluding the end mark from the constraint we restrict the exploration to those executions where the flash occurs only after the second incoming call has arrived.

In the interval between the arrival of the second incoming call and the flash, or between any two adjacent marks, there may be zero or more unspecified events. If there are zero intervening events, the flash is the very next execution step after the arrival of the second incoming call.

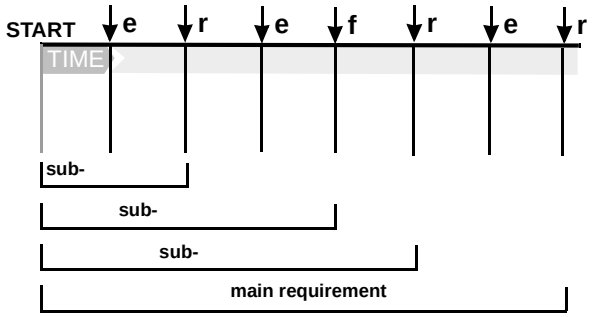


Figure 7. There is a sub-requirement for each required event included in a requirement.

A system constraint can be more complicated condition than a simple event. The example requirement in Figure 4, for instance, uses the condition idle in a constraint. Idle is not an event but rather expresses a condition on the system state, which is defined separately in terms of, e.g., values of variables at the state. Idle is a fairly general condition that includes both execution sequences where the user has never gone offhook and those in which the user has gone onhook since the last offhook. In our system, Idle requires that there are no active or held parties.

A single timeline can contain multiple regular, required and fail events. Each required or fail event in essence defines a sub-requirement, as shown in Figure 7. If any sub-requirement is violated, that is if any required event in the specification does not occur while its preamble does, or if any fail event occurs after its preamble, it is an error.

In general, a timeline must contain at least one required or fail event. Normally, the final event on the timeline will be a required or fail event. A timeline need not contain constraints.

5. Timelines as test automata

In this and the next section we describe how a timeline specification is to be converted mechanically into an equivalent test automaton that can be used in a logic model checking process, e.g., with the model checker Spin [5]. This description will also serve as our formal semantics for what a timeline really means. A more formal description of the translation is provided in the appendix as pseudocode.

The test automaton produced from the timeline specification is a kind of automaton called a Büchi automaton, as will be explained shortly. Consider this requirement from the LSSGR for the Call Waiting (CW) feature:

CW tone should be applied to the called party as an indication of a waiting call. It may be applied twice to the line with CW, once when the incoming call arrives, and then again approximately 10 seconds later if the CW line has not yet responded to the CW call.

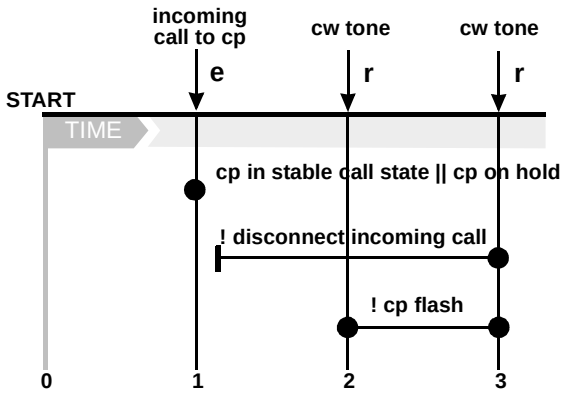


Figure 8. Timeline for a Call Waiting requirement

The corresponding timeline for this requirement is shown in Figure 8. This timeline states that when the call waiting subscriber (the cp) receives an incoming call, the call waiting tone (cw tone) should be given twice. The subscriber response that is awaited is a flash, so the timeline is constrained so that no calling party flash (! cp flash) occurs between the first and second occurrence of the call waiting tone. A flowchart diagram that appears at the end of the requirement document further illustrates that if the incoming call disconnects no further call waiting tones should be given. Therefore, the constraint that the incoming call does not disconnect (! disconnect incoming call) also applies. For this requirement to apply, the CW subscriber must be in a stable call state or on hold when the incoming call arrives. The subscriber is said to be in a stable call state if no change to the current connection is anticipated. The subscriber is said to be 'on hold' if another party has placed the subscriber on hold.

Interpreting this timeline, the TimeLine Editor generates the test automaton illustrated in Figure 9. Event names are replaced by a propositional symbol, in this case the symbols **p1** through **p5** shown in the KEY of Figure 9. The states of the automaton are the nodes in the graph, represented by circles. The events, represented by arrows, drive the transitions between states.

When the check begins we start out in state **s0** in the test automaton. At each step in the execution of the system a transition in the test automaton is made. As long as event **p1** together with constraint **p3** does not occur, the automaton remains in its initial state, by traversing the self-loop on state **s0** at each execution step of the system. If and when **p1** and **p3** occurs, the test automaton can move to state **s1**. State **s1** is called an *accepting state*, indicated by the double circle. If we find an execution where the test automaton can remain in such an accepting state

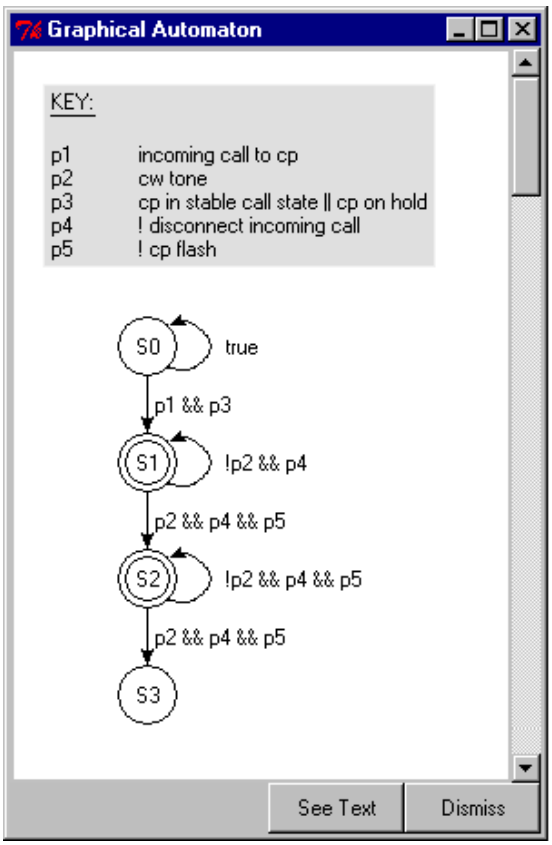


Figure 9. An automaton for timeline in Figure 8

indefinitely, that execution constitutes a violation of our timeline requirement, and the model checker will report it as an error. If, however, we see a **p2** together with the corresponding constraints **p4** and **p5**, the test automaton moves to state **s2**. State **s2** is also marked as an accepting state, which means it is an error if a system execution causes the test automaton to remain in this state indefinitely. So, another occurrence of, **p2 && p4 && p5,** must be observed to avoid an error report from the model checker.

The interpretation of *accepting states* is what differentiates a Büchi automaton from a standard finite automaton. A Büchi automaton is used to trap infinite execution sequences that violate a requirement, whereas a standard automaton can only trap finite execution sequences. The graphical test automaton in Figure 9 is produced to aid the user in visually inspecting the test automata. The Spin model checker uses the *Never Claim* version of the automata, also produced by the TimeLine Editor, and depicted in Figure 10.

6. Mechanical conversion of timelines to automata

The TimeLine Editor converts timeline specifications to test automata using a straight-forward algorithm. The

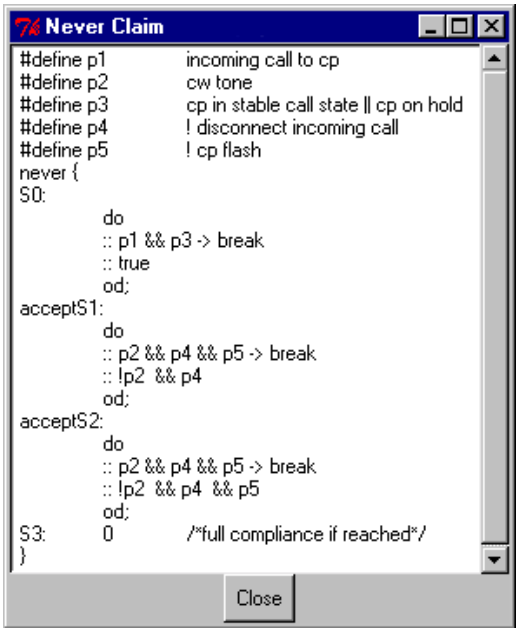


Figure 10. Never claim for automaton in Figure 9

number of states we will need in the test automaton is equal to the number of events on the timeline plus one. In the case of the Call Waiting specification shown in Figure 9, there are three events, hence four states in the automaton. In each state we are waiting for the next successive event on the timeline. For instance, for the automaton in Figure 9 and Figure 10, we start out in state **s0** and we are waiting for **p1** (incoming call). While we wait for **p1** we may detect other events (outgoing call, flash, onhook, etc.) and on each of these events we traverse the self loop **true**. In state **s1** we wait for **p2** and in state **s2** we wait for the second instance of **p2.** If we just consider the events on the timeline specification and ignore the constraints for now, we derive the test automata shown in Figure 11.

Both **s1** and **s2** have a double circle, indicating that they are special states, called *accepting states*. These states are *accepting* because if we can remain in these states indefinitely for a particular system execution under consideration, the execution is flagged as an error by the model checker. Hence, for this example, if we can remain in state **s1** indefinitely, waiting for the first required call waiting tone, this execution will be flagged as an error.

Likewise, if we can remain in state **s2** indefinitely, waiting for the second required call waiting tone, this will also be an error. In general, each required event will have an associated accepting state where we wait for that event to occur. The accepting state for a required event labeled **r** will have a self loop labeled **!r**. The states, like **s0**, associated with regular events, **p1** in the case of **s0**, will also have a self loop labeled **true**.

Once the events have been used to form the structure of the automata, and the event types (normal versus required)

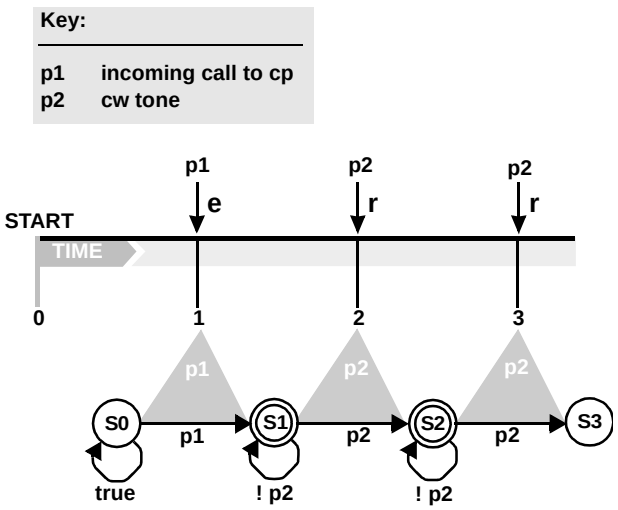


Figure 11. Test automata for timeline specification shown in Figure 8, including events only

have been used to identify normal and accepting states and transition labels, we can add the constraints. To do this we first construct a list of the constraints that overlap each event. These are summarized in Table 1.

Table 1. Constraints overlapping events for timeline in Figure 8

Event	at Marker	Constraints
p1	1	p3
p2	2	p4 & p5
p2	3	p4 & p5

Table 2. Constraints that apply between events for the timeline in Figure 8

Between Events	Between Markers	Constraints
p1 & p2	1 & 2	p4
p2 & p2	2 & 3	p4 & p5

Overlapping constraints are added via conjunction to the label of the transition coming out of the *waiting state* associated with an event. Thus, since **p4** and **p5** overlap event **p2** at mark 2, **p4 && p5** is added to the transition labeled **p2** out of state **s1**.

In addition, a constraint that applies in the interval immediately prior to an event labeled **e** and subsequent to the event preceding **e** (or starting at the initial **START** mark if **e** is the first event) on the timeline is added by conjunction to the self loop of the *waiting state* of event **e**. Constraints that apply in the intervals between events are summarized in Table 2.

In the case of event **p2** at mark 2, the constraint **p4** applies in the interval between **p1** at mark 1 and **p2** at mark 2, so the constraint **p4** is added to the self loop at

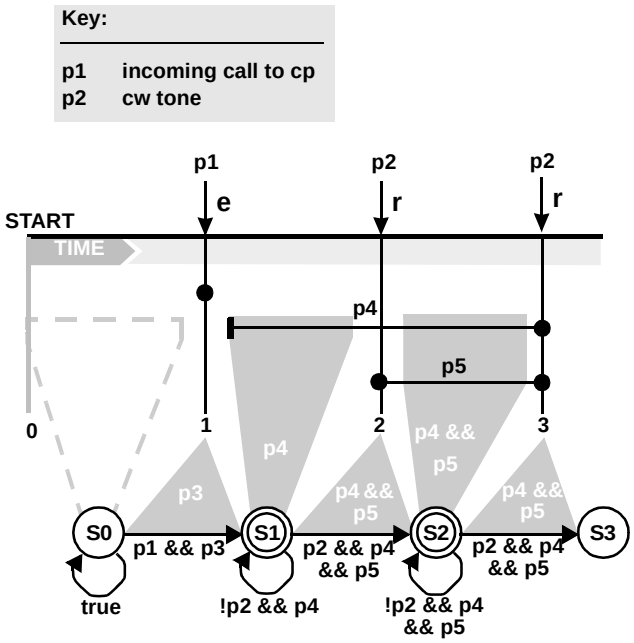


Figure 12. Test automata for timeline specification shown in Figure 8, including events and constraints

state **s1**. Using the algorithm outlined above, we generate the final test automaton shown in Figure 12.

So far our description of automated automata generation has not addressed fail events. First we will discuss the general case of a fail event, that is, a fail event that is an intermediate event. Then we will discuss a special case, when the fail event is the last event.

We will define the *progress path* of a generated automaton to be the path consisting of those states where we wait for required or regular events and those transitions that we take upon the reception of required or regular events. States in the progress path are labeled **S{N}**, where **N** is the progress state number. Transitions in the progress path are labelled with the normal and required events and constraints that apply, as described previously. An automata generated by the Timeline Editor will contain a single progress path and one fail path corresponding to each fail event in the timeline. A fail path is a path leading out of the progress path and terminating in an accepting state, called a fail state. Fail states are labeled **F{N}** where **N** is the fail state number. Hence, the automata for a timeline with no fail events will contain only a progress path, and an automata for a timeline with **N** fail events will contain a progress path and **N** fail paths.

Each fail event will have an associated accepting state that we transition to if the fail event occurs in the specified interval. The transition to the fail state is made from the wait state associated with the regular or required event directly following the fail event. So for the requirement in

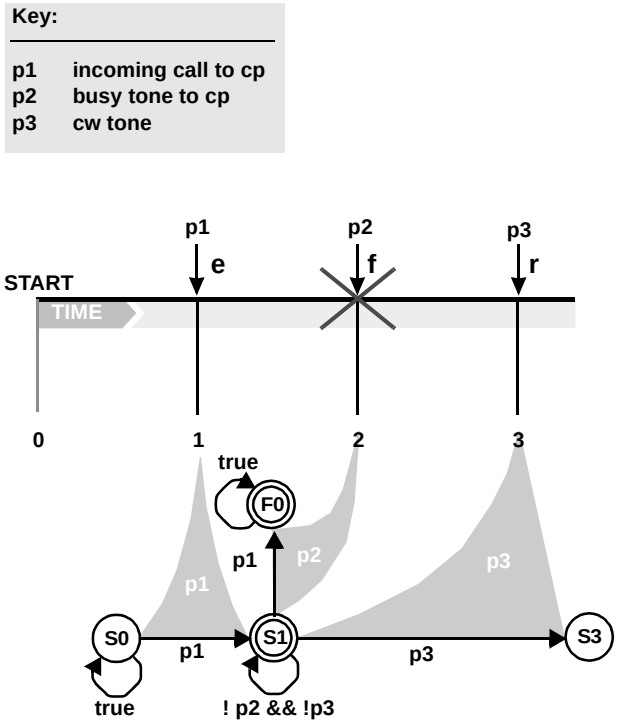


Figure 13. Test automata for timeline containing intermediate fail event, including events only

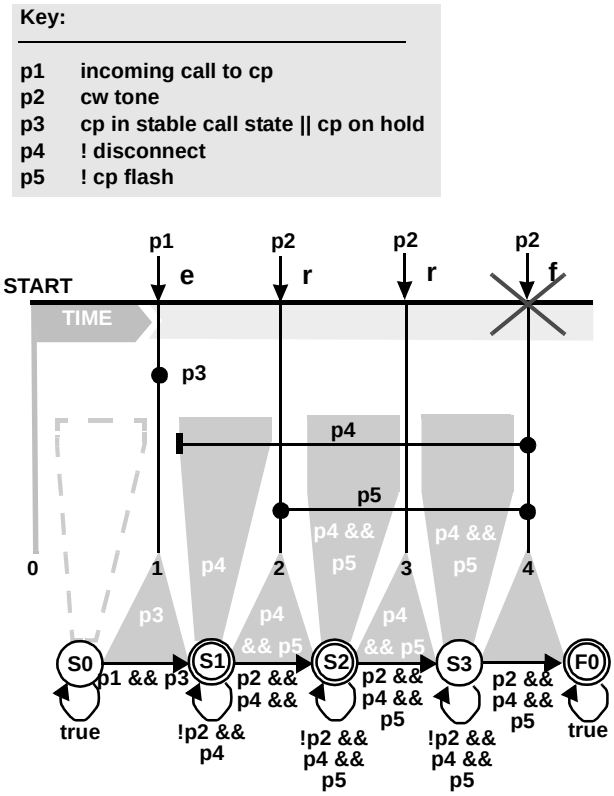


Figure 14. The call waiting requirement in Figure 8 is strengthened by adding a third call waiting tone as a fail event at the end of the timeline

Figure 13, the transition to the fail state **F0** is made from the wait state associated with required event **p3** (**cw tone**). The fail state has a self loop labeled **true** because once in the fail state, we remain there for the remainder of the execution.

Constraints that apply on the transition to the fail state are added via conjunction to the fail transition. Determining which constraints apply on the transition to the fail state is done in the same manner as for regular and required events; by inspecting which constraints overlap the fail event and which constraints apply between the mark preceding the fail event and the mark to which the fail event is attached.

A special case is a timeline that has a fail event as the last event, as in Figure 14. This variation on the call waiting requirement states that it is an error if a third call waiting tone is given. The automaton for a timeline with a fail event as the last event is created by a adding a fail transition from the last progress state to a fail state. Any constraints that overlap the mark to which the final fail event is attached are added via conjunction to the fail transition.

Timelines are restricted to have at most one consecutive fail event. To express that more than one event can cause a transition to the fail state while waiting for the next regular or required event, the names of the fail events may be joined via a logical or (**||**) on a single fail event label. Constraints must be contiguous between non-fail events.

7. Expressiveness of timelines

Test automata generated from timeline specifications constitute a limited fragment of the properties expressible in Linear Temporal Logic. In particular, timelines without fail events correspond to certain liveness properties. The formal definition of *liveness* [8],[10] requires that any finite system execution must be extendable into an infinite execution that satisfies the given requirement (i.e., that does not produce a violation). From any state in the automaton we can build a finite sequence of events that leads us to the final rejecting state (state **s3** in Figure 9). This means that every timeline that generates such an automaton satisfies the definition of a *liveness* requirement. It can also be shown that a timeline with *k+1* events minimally requires an LTL formula with a, so-called, *Until-depth* of *k* [2]. This means that we would have to use *k* nested formulas to represent the same requirement, which makes the LTL requirement hard to read if more than two or three events are used in sequence.

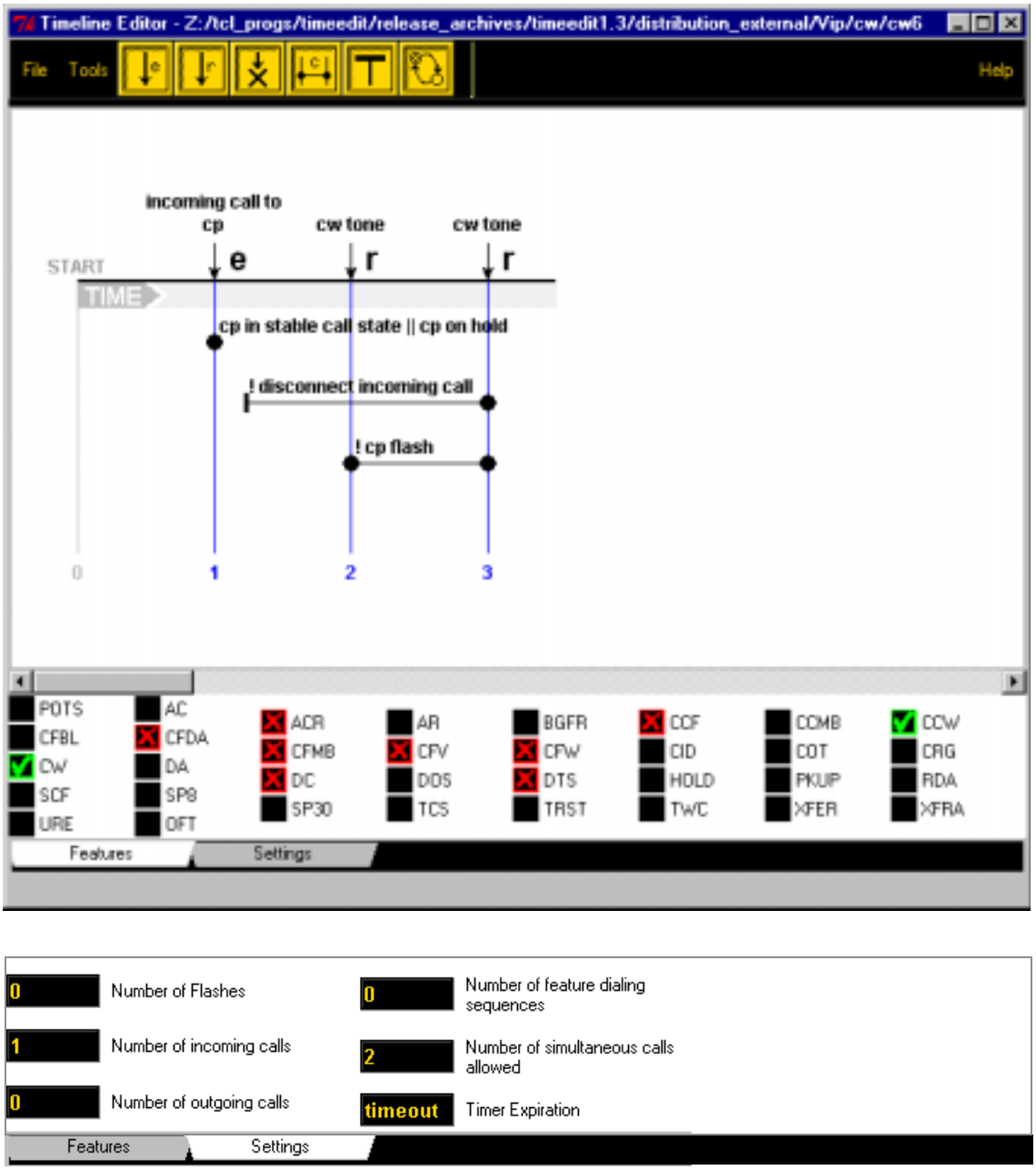


Figure 15. Timeline Editor tool interface

When fail events are present, we can express more than just liveness properties. For example we can express the simple safely property that a particular (fail) event should never occur.

Model checkers such as Spin can optimize the verification process if it can be guaranteed that correctness requirements are *stutter-invariant* [3], meaning that the requirement is not sensitive to stuttered, or repeated, individual events. Requirements expressed in the subset of LTL without a next operator, for instance, have this desirable requirement. Stutter-invariance cannot be guaranteed, though, for timeline specifications. However, fairly simple algorithmic checks on the generated automata can be used to determine whether or not a timeline requirement is stutter-invariant [4], [11], so that the verification process can be adjusted accordingly.

Timeline specifications do not express real-time or performance requirements. Hence, on the few occasions

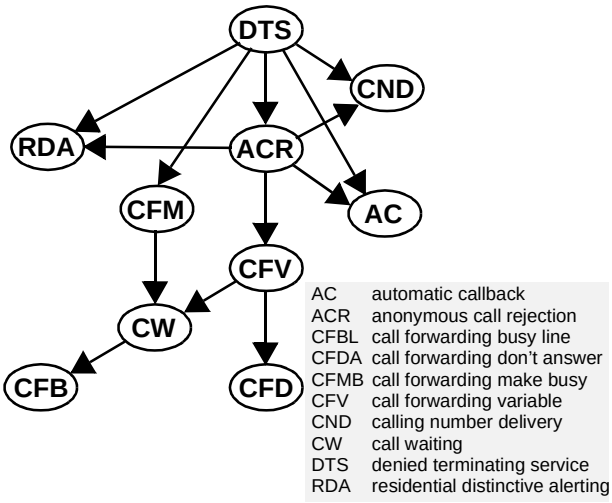


Figure 16. Precedence relations for features that are triggered by the arrival of an incoming call

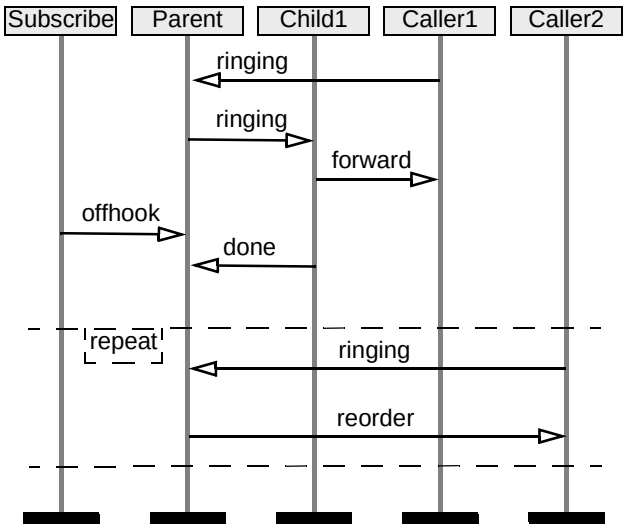


Figure 17. An example requirement violation

that one of these requirements was encountered in the standards, such as the Call Waiting requirement that the second call waiting tone must be given within 10 seconds of the first call waiting tone, we did not test it.

8. Global constraints

In addition to constraints on events, we also occasionally need to specify constraints of a more general nature. We call these *global constraints*.

Global constraints depend on the type of system that is being checked. For a telephone switch, the set of features that has been provisioned for a given subscriber form a global constraint. Certain subsets of features are triggered by the same events and for these there is generally a precedence relation in the feature standards that defines which feature should be invoked when all are provisioned for a given subscriber. For instance, several features could be triggered by the arrival of an incoming call. Figure 16 shows the precedence relations for features that are triggered by the arrival of an incoming call, where higher precedence features point to lower precedence features.

If we want to test a requirement for the *Call Waiting* feature, we will need to: *enable* CW, and *disable* the higher precedence features, otherwise we will not be able to consider the executions where *Call Waiting* is invoked. We do, however, want to explore both the *enabled* and the *disabled* cases of potentially conflicting lower precedence features (e.g., CFBL) and potentially conflicting features that are not in *Call Waiting's* precedence hierarchy (there are none in this case), to ensure that these features are not mistakenly invoked when *Call Waiting* is enabled

The Timeline Editor allows the user to select from available global constraints using the interface shown in Figure 15. Under the **Features** heading the user can require that the feature be disabled (an 'X'), enabled (a check mark), or that the model checker should consider both cases (indicated by a blank selection box).

The **Settings** field defines additional global constraints that control the behavior of our test harness for verifying telephony software. The test harness is composed of abstract models for device behavior, subscriber behavior, and timers. These environment models are purposely designed to be hostile to further increase the scope of the model checking process. For instance, if the subscriber can generate offhook, onhook, digit, and hook-flash events, etc., the subscriber model will assume that the subscriber can generate these for even an erratically behaving subscriber. Using this approach, the model checker can verify that under no circumstances will it be possible for even an erratically behaving subscriber to cause errors in the behavior of the telephone switch.

The **Settings** field can be used to fine tune the test harness behavior by stipulating, for instance, that exactly two flashes should be generated by the subscriber, that there should at most 1 incoming call, that the a timer may expire only if no other events can be processed, etc.

9. Error traces -- an example

An error trace found by the model checker can be displayed as either a message sequence chart or a series of concurrent execution steps, interleaved in time. Error traces reported by the model checker are often sequences with subtle race conditions, leading to a fault.

Such is the case with one violation of the dialtone requirement from Figure 3. A violation of the requirement, displayed by the model checker as the message sequence chart in Figure 17, occurs if the subscriber has call forwarding and happens to pick up the phone precisely when an incoming call is being forwarded. The call processing software can delay the generation of dialtone arbitrarily long while the system is rejecting or forwarding a stream of incoming calls. When the calls stop, the system will eventually time out and deliver dialtone (not shown here).

The Parent and Child processes in the message sequence chart are system software components, whereas processes Subscriber, Caller1, and Caller2 are test stubs. In the software architecture of the system we tested, the Parent process spawns a Child process for each new call in which the logical subscriber participates.

The model checker can also present an error scenario as a trace of C statement executions, so that the developer can analyze the sequence of executed statements leading to the error. A scenario such as this one can be extremely hard to detect with normal testing techniques, yet fairly trivial to generate with the help of a logic model checking tool

10. Conclusions

.The TimeLine Editor can simplify and speed-up the capture of formal requirements to be used in both testing and formal model checking.

The TimeLine Editor software may be downloaded at:

http://www.bell-labs.com/topic/swdist/

Implementing the TimeLine Editor took about one month, after which we quickly used the tool to express 117 requirements in two months time, which included analysis of copious standards documents.

Of the 117 requirements we specified using the TimeLine editor, the average timeline specification contained 4 to 5 events, and 2 to 3 constraints. The most complex timeline specification contained 11 events and 7 constraints, and the simplest contained 2 events and one constraint. Thirty-eight percent of the events were required events, and the remainder served to provide context for the requirement.

The TimeLine Editor is one line of work in an ongoing effort to automate more aspects of the formal verification process of complex software that we are pursuing. The goal of the automation is to hide what the model checker does from the user so that the user does not need special training in logic to exploit the power of model checking technology in systems verification. Ultimately, we would like be able to generate testable requirements directly from a machine readable requirements model.

Most recently, we have integrated the TimeLine Editor into our new FeaVer [6] front-end tool, giving the user the ability to formulate and run tests interactively from a single interface. There are still many ways in which we may be able to improve the usefulness of our TimeLine Editor. For instance, we can extend the tool by supporting a graphical method for defining *coregions* of adjacent events on the timeline, to define groups of events that may occur in arbitrary order, rather than in strict timeline order. Or, we can link the TimeLine Editor more directly to the model checker, to provide feedback about reachable and unreachable portions of the timeline specification. This would give the user visual feedback on whether or not the preamble is correctly stated.

11. Reference

[1] L. K. Dillon, G. Kutty, L.E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A Graphical Interval Logic for Specifying Concurrent Systems. *ACM Trans. on Software Engineering and Methodology*, 3(2), pp. 131-165, 1994.

[2] K. Etessami and T. Wilke, An Until hierarchy for temporal logic. *11th Ann. IEEE Symp. on Logic in Computer Science.* 1996, pp. 108-117.

[3] G.J. Holzmann and D. Peled, An improvement in formal verification. *Proc. Formal Description Techniques*, Forte94, Berne, Sw., Chapman&Hall, 1994, pp. 197-211.

[4] G.J. Holzmann and O. Kupferman, Not checking for closure under stuttering, In: *The Spin Verification System*, American Mathematical Society, 1996, pp. 17-22.

[5] G.J. Holzmann, The model checker Spin, *IEEE Trans. on Software Eng.*, 5(23):279-295, 1997.

[6] G.J. Holzmann and M.H. Smith, Automating software feature verification, *Bell Labs Technical Journal*, 5(2):72-87, 2000.

[7] http://www.cis.ksu.edu/santos/spec-patterns/

[8] L. Lamport, Proving the correctness of multiprocess programs. *IEEE Trans. on Software Eng.*, 3(2):125-143,1977.

[9] LSSGR, *LATA Switching Systems Generic Requirements*, FR-NWT-000064, 1992 Edition. Feature requirements, including SPCS capabilities and features. SR-504, Iss. 1, March 1996, Telcordia/Bellcore.

[10]Z. Manna, and A. Pnueli, *The temporal logic of reactive and concurrent systems*, Vol. 1, Springer-Verlag, 1992.

[11]D. Peled, T. Wilke, and P. Wolper, An algorithmic approach for checking closure requirements of temporal logic specifications and w-regular languages. *Theoretical Computer Sciences*, 195(2):183-203, March, 1998.

[12]A. Pnueli, The temporal logic of programs. *Proc. 18th IEEE Symposium on Foundations of Computer Science*, 1977, Providence, R.I., pp. 46-57.

[13]R. Schlor, and W. Damm. "Specification of System-Level Hardware Designs Using Timing Diagrams". In Proc. European Design Automation and European Event in ASIC Design, Feb. 1993, IEEE Press, pp. 518-524.

Appendix

The pseudocode that follows provides a description of the translation of a timeline to a Büchi automata. The algorithm keeps track of a set of states state_set, and a set of labeled transitions between them called transition_set. Each state in the state set has an associated flag indicating whether or not it is an accepting state. We process the timeline one event at a time, from left to right, associating the variable Event with the current event under consideration. The following convention is used in the pseudocode notation below:

C: denotes the conjunction of the labels of those constraints that apply to the interval between Event and the previous event (immediately prior to Event).

C': conjunction of the labels of those constraints that overlap Event.

pp: denotes the current "progress point" on the "progress path", which is used as a marker during the translation.

We note that the automaton generated by the pseudocode below actually omits the extra redundant non-accepting state at the end of the progress path that has been depicted in the timelines throughout the paper. The exclusion does not affect the meaning of the Büchi automaton.

Pseudocode:

```
initialize:
   state_set = {init} /* intial state set */
   transition_set = {}
   pp = init /* the initial progress point */
while (more events) {
   Event = get_next_event()
   if (is_Fail(Event)) {
      if (pp = init) {
         add transition (pp, TRUE && C, pp)
         /* add to transition_set */
      } else {
         add transition (pp, !Event && C, pp)
      }
      add new accept state v /*to state_set*/
      add transition (pp, C', v)
      add transition (v, TRUE, v)
   }
   if (is_Required(Event)) {
      make pp an accept state
      add transition (pp, !Event && C, pp)
      add new non-accept state v_Event
      add transition (pp, Event && c',v_Event)
      pp = v_Event
   }
   if (is_Regular(Event)) {
      if (pp = init) {
         add transition (pp, TRUE && C, pp)
      } else {
         add transition (pp, C && !Event, pp)
      }
      add new state v_Event
      add transition (pp, C' && Event, v_Event)
      pp = v_Event
   }
}
```