Deriving Manuals from Formal Specifications
—
Extended Version

M. Massink[1] and D. Latella[1]

[1] Istituto C.N.R.-ISTI,
Via Moruzzi 1, Pisa, Italy
email: {m.massink,d.latella}@cnuce.cnr.it

Abstract. In this paper we propose the use (or re-use) of formal specifications for the development of user manuals and user instructions. We base our work on two observations. The first is that a considerable part of user manuals consist of series of user instructions that guide a user in the use of a device or a software application. The second is that the user interface in terms of user operations and feedback can often be described in the form of a finite state machine. In this paper we describe how model checking techniques can be used to derive automatically series of user instructions from the interface specification. The approach is illustrated by means of the specification of a well-known device such as a telephone which is then used to derive proper instruction sequences addressing relevant 'how-to' and 'what-if' questions about the operation of the system from a user point of view.

Keywords: D.2.2. - Design Tools and Techniques, D.2.4. - Model Checking, D.2.7. - Documentation.

1 Introduction

The production of good user manuals and documentation of devices and software applications is not an easy task and leads in many cases to difficulties in the use of those artifacts by the end-users. This has been very well illustrated in several books (see e.g. [8]). The use of models of the interface, in particular models that specify the operations that a user can perform, the effect on the system and the corresponding feedback on these effects that the user may expect have facilitated the analysis of the usability of an interface extent [11, 4].

In this paper we propose to use and, where possible, to re-use specifications of the user interface for the derivation of series of user instructions for user manuals. Specifications of the user interface can often be presented in some form of finite state machine, transition matrix or labelled transition system that relates user operations to changes in system state and feedback to the user. In that form they can also serve as an input model for model checkers [2]. Model checkers are software tools that can perform behavioural analysis of formal models by checking the behaviour described by the model against a temporal logic formula (LTL) [9]. The result of this analysis is a yes/no answer indicating whether the model satisfies the logic formula, but also an example trace that shows how it was possible that a formula was not satisfied.

This last feature of model checkers may be used in a somewhat indirect way to obtain series of user instructions from interface models by means of model checking. Of course, we need to express the proper temporal logic formulas in order to obtain traces that are indeed the series of user instructions that we are looking for.

Generating documentation based on formal specifications has been proposed in the context of several different approaches. For example, in the IDAS project [10] the aim was to derive technical documentation in readable natural language. In this paper we do not deal with natural language generation, though such an approach could could be very well used to enhance the results obtained. In [12] logical specifications in TLA+ are automatically generated from annotated state machines which have been obtained by re-engineering existing manual fragments. The logical specification is then used to generate a logical manual. Consistency and completeness checks have been performed by hand. The use of model checkers, as we propose in this paper, could potentially greatly enhance the approach in [12] because it allows for automatic verification of correctness properties, beside the automatic generation of series of user instructions. The

modelling language of model checkers allow for the specification of interactively complex and concurrent systems for which the development of proper manuals is quite a complex issue. In this paper we illustrate the basic principles of how a model can be used to generate series of user instructions. Fully automatic generation of complete manuals would require a further synthesis of this basic idea with proper tools for natural language generation [10, 6] or on-line (interactive) helpers such as e.g. [3].

The paper is organised in the following way. Section 2 describes the case study of a plain old telephone system that will serve as a running example throuhout the paper. Section 3 gives a formal specification of the user interface of the telephone in the input language of the SPIN model checker. Section 4 addresses the problem of translating the user-oriented 'How to' and 'What if' questions into suitable temporal logic formulas and illustrates the outcomes of several of those questions when given to the SPIN model checker. In Section 5 we draw some conclusions and discuss how the approach could be improved to obtain more readily usable results.

2 POTS Phone Manual Case Study

In this paper we illustrate our approach to the derivation of user instructions by means of a simplified case study concerning the well-known plain old telephone system (POTS). This system has a hook and a sensor that signals when the phone is on-hook or off-hook. The hook also serves for voice communication in two directions and gives the user audible signals to indicate particular situations that may occur when the phone is used, such as a dial-tone that indicates that the user may start dialing a number, a busy tone etc. Furthermore, the phone has a number of buttons to dial phone numbers.

2.1 A Plain Old Telephone Service (POTS) system

The phone system we consider is a simple version of the common fixed phone set that can be found in many households. At the user interface level the phone can be described as a rather simple finite state machine with 12 states, 3 user operations and 7 product responses. The user operations and product responses are shown in Table 1. The states are presented directly in the transition system of Fig. 1. The initial state of the phone is given by two concentric circles labelled by STANDBY. In that situation the phone is on-hook and waiting for an incoming phonecall or a user operation. The user is notified of an incoming phonecall by a ringing signal produced by the phone. When the phone is ringing there are two possibilities. The user can pick up the phone and start the voice communication, or, when the user is waiting too long, the phone stops ringing and returns into standby being silent (no response). In the specification we have abstracted from time to keep the specification simple, so the time-out that stops the ringing signal is not explicitely modelled.

When the user is speaking at the phone there are again two possibilities: the user decides to end the conversation and put the hook back on the phone (gon and ONHOOK), or the other conversation partner decides to end the conversation first and the user hears the no-connection tone indicating that the other user is disconnected. Also in this case the user needs to put the hook back onto the phone and the phone returns to the STANDBY mode.

When the user takes the initiative to make a phonecall, (s)he needs to take the phone off-hook and hears a dial-tone (DTN). There are again several possibilities. The usual course of actions will be that the user dials the number (d), waits for the connection to be established and starts voice communication (VCE). Several other things may happen instead. The user may decide to interrupt the calling procedure and put the hook back on the phone instead of dialing a number. The user may also wait too long before dialing the number and the signal given by the phone (ROH) indicates the receiver off-hook tone. In that case the user needs to replace the hook on the phone before any other operations can be performed.

Also after dialing the number several things may happen. The line may be occupied, and the user hears a busy signal (BSY) and needs to replace the hook on the phone. It may also take too long before the conversation partner answers the phone and a receiver off-hook signal is presented. Again, the user needs to place the hook back on the phone before (s)he can proceed with further operations.

The transition system shown in Fig. 1 shows the states and the transitions modelled. The transitions are labelled by the user operations and device responses. Those marked by an exclamation mark (!) are

User operations		Product responses	
gof	go off-hook	DTN	present dial tone
gon	go on-hook	VCE	carry voice conversation
d	dial number	BSY	present busy tone
		ROH	present receiver off-hook tone
		NRS	no response
		NCT	present no connection tone
		ring	present ringing tone

Table 1. User operations and product responses

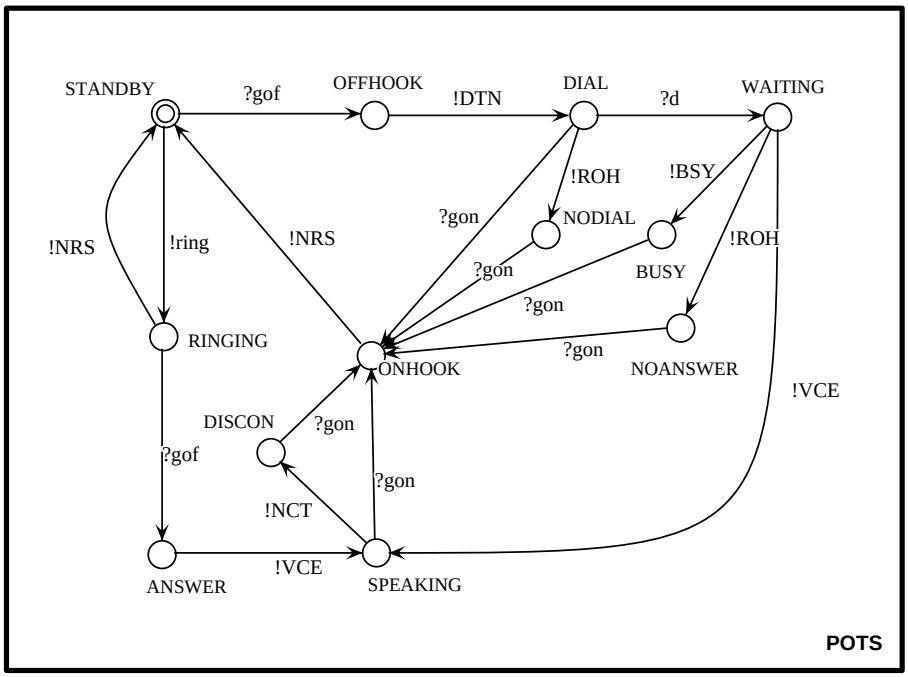

Fig. 1. LTS specification of the phone interface

generated by the phone, those with a question mark are operations that the phone may accept on initiative of the user.

3 Formal Model of the User Interface

So far we have developed a rather precise example specification of the somewhat simplified behaviour of the interface of a phone at the user level of abstraction. Such a model may be obtained by abstraction from more detailed formal specifications of the phone system that are intended to describe the internal operation of the system.

In this section we translate the model into a more formal one that can be used as input to a model checker. We chose to use the model checker SPIN [7] for this article because it is one of the widely used open source model checkers available for different platforms and with a graphical user interface. However, our approach is largely independent of the particular model checker chosen.

A model checker takes a formal model of the behaviour of a system and a temporal logic formula and verifies automatically whether the formula is satisfied by the model by exhaustive state-space exploration. When the formula is not satisfied, the model checker generates a counter example that shows a sequence

of events that leads to the violation of the property. We refer to appendix A for a short overview of linear temporal logic formulas used in this article.

We proceed in the following way. First we perform a straightforward translation of the POTS phone transition system into the format of Promela, the modelling language of SPIN. In this case the translation has been performed by hand, but it could in principle be automatized. The POTS phone is modelled as a Promela process called POTS. Furthermore, we modelled another process, user, that interacts with the POTS phone by accepting or initiating any of the possible interactions with the phone. These could be understood as the set of basic actions that the user is familiar with when operating a phone, though we assume that the user does not know in which order they (need to) appear when the user tries to use the phone having higher level goals in mind such as making a phone call.

In a second stage we formalize the higher level goals of the user in the form of ‘how to’ and ‘what if’ questions formulated as particular forms of temporal logic formulas. The Promela specification and the logic formulas together form the input that is given to the model checker such that it can produce the user instructions automatically.

Following is the Promela specification of the phone system and the generic user. It contains two processes, one modelling the POTS phone and one modelling the generic user. The POTS process reflects the structure of the labelled transition system of Fig. 1 where each state in the LTS is represented as a label in Promela and the possible transitions in the choice structure following the label. The interaction between the user and the POTS phone is modelled by a shared channel called ‘a’ for synchrononous (rendez-vous) communication.

The two processes are run in an atomic way in parallel, generating all possible states that would result from any interleaving execution. The atomic statement ensures that both processes are started together.

```
mtype = {gof,gon,ring,d,NRS,VCE,NCT,DTN,ROH,BSY};
chan a = [0] of {mtype};

proctype POTS (chan u)
{
STANDBY:        if
                :: u?gof -> goto OFFHOOK
                :: u!ring -> goto RINGING
                fi;

RINGING:        if
                :: u!NRS -> goto STANDBY
                :: u?gof -> goto ANSWER
                fi;

ANSWER:         u!VCE; goto SPEAKING;

SPEAKING:       if
                :: u?gon -> goto ONHOOK
                :: u!NCT -> goto DISCON
                fi;

ONHOOK:         u!NRS; goto STANDBY;

DISCON:         u?gon; goto ONHOOK;

OFFHOOK:        u!DTN; goto DIAL;

DIAL:           if
                :: u?d -> goto WAITING
                :: u!ROH -> goto NODIAL
                :: u?gon -> goto ONHOOK
                fi;
```

```
WAITING:        if
                :: u!BSY -> goto BUSY
                :: u!VCE -> goto SPEAKING
                :: u!ROH -> goto NOANSWER
                fi;

NODIAL:         u?gon; goto ONHOOK;

BUSY:           u?gon; goto ONHOOK;

NOANSWER:       u?gon; goto ONHOOK
}

proctype user (chan u)
{
        do
        :: u?ring
        :: u?NRS
        :: u?VCE
        :: u?NCT
        :: u?DTN
        :: u?ROH
        :: u?BSY
        :: u!gof
        :: u!gon
        :: u!d
        od
}

init
{atomic {run user (a); run POTS (a)}}
```

4 ‘How to’ and ‘What if’ questions

A considerable part of user manuals consist of descriptions of how certain functionality of a device or a system can be activated. For example in a manual of a video recorder we find series of instructions on how to set the timer, tune the TV channels, various ways to program the recording, how to play back the tape and so on and so forth. Another part consists of the description of the basic interface points of the device (buttons, jacks, displays) and a list of problem shooting items for when unexpected results are encountered.

Most users will consult a manual as a reference document rather than reading it from the beginning to the end. Often they will consult the manual with one of the following kinds of questions: ‘How to’ questions and ‘What if’ questions. The first refer to the use of certain functionality that users expect the device to provide, and they would like to know how this functionality can be activated. The second kind of questions may occur in situations where the device behaves different than expected, and the user needs to know how to continue from such a situation to reach a more satisfactory one. Often manuals are more explicit about the ‘How to’ than the ‘What if’.

In this section we shall address both kinds of questions, refine them into more detailed and concrete questions, and show how they may be formulated into apropriate LTL formulas. The aim is to arrive at certain recurring patterns of such formulas that may be useful in a more general way.

How to make a phone call?

The first question we address is “How to make a phonecall”. This corresponds to asking how, starting from the initial state, a state can be reached in which the user is speaking to someone else by phone. In terms of the Promela model we would like to obtain a sequence of instructions that guide the user from the initial state (the phone is silent and on hook) to a state in which a voice conversation is going on. This last state is characterised by process POTS being in state “SPEAKING”.

SPIN provides automatically example traces when a universal property does *not* hold. So in order to obtain a trace, we should not verify the formula 'eventually p holds' ($<>p$), where p stands for the phone being in the state SPEAKING, but rather its negation: $!<>p$. In this way we 'provoke' the model checker to verify that from the initial state we will never eventually be able to reach a state in which the user is speaking by phone. Of course, the model checker will immediately give us a counter example showing that the SPEAKING state *can* be reached and this is what we were up to. Note that the formula $!<>p$ is equivalent to $[]!p$, this can be shown by a simple formal proof. In order to get a usually better result, we may instruct SPIN to look for the *shortest* trace leading to the state SPEAKING. This is useful for obtaining the shortest set of instructions for the user.

In SPIN we need to define the exact state that we are looking for. In this case we require that process POTS is in state SPEAKING. [1] The formula is the one explained above.

```
#define p (POTS[2]@SPEAKING)
[]!p
```

When instructing SPIN to produce a shortest trace that violates the property, it produces the following detailed trail:

```
preparing trail, please wait...done
spin: couldn't find claim (ignored)
  2:    proc  0 (:init:) line  65 "pan_in" (state 1)    [(run user(a))]
  3:    proc  0 (:init:) line  65 "pan_in" (state 2)    [(run POTS(a))]
  5:    proc  2 (POTS) line   8 "pan_in" (state -)      [values: 1!ring]
  5:    proc  2 (POTS) line   8 "pan_in" (state 3)      [u!ring]
  6:    proc  1 (user) line  51 "pan_in" (state -)      [values: 1?ring]
  6:    proc  1 (user) line  51 "pan_in" (state 1)      [u?ring]
  8:    proc  1 (user) line  58 "pan_in" (state -)      [values: 1!gof]
  8:    proc  1 (user) line  58 "pan_in" (state 8)      [u!gof]
  9:    proc  2 (POTS) line  13 "pan_in" (state -)      [values: 1?gof]
  9:    proc  2 (POTS) line  13 "pan_in" (state 9)      [u?gof]
 11:    proc  2 (POTS) line  16 "pan_in" (state -)      [values: 1!VCE]
 11:    proc  2 (POTS) line  16 "pan_in" (state 13      [u!VCE]
 12:    proc  1 (user) line  53 "pan_in" (state -)      [values: 1?VCE]
 12:    proc  1 (user) line  53 "pan_in" (state 3)      [u?VCE]
 14:    proc  2 (POTS) line  20 "pan_in" (state -)      [values: 1!NCT]
 14:    proc  2 (POTS) line  20 "pan_in" (state 17)     [u!NCT]
 15:    proc  1 (user) line  54 "pan_in" (state -)      [values: 1?NCT]
 15:    proc  1 (user) line  54 "pan_in" (state 4)      [u?NCT]
spin: trail ends after 16 steps
#processes: 3
 16:    proc  2 (POTS) line  25 "pan_in" (state 23)
16:    proc  1 (user) line  50 "pan_in" (state 11)
 16:    proc  0 (:init:) line  65 "pan_in" (state 4)
3 processes created
Exit-Status 0
```

This trace is visualised in the message sequence chart, also produced by SPIN, shown in Fig. 2:

The trace shows that the shortest way (i.e. least number of user-operations) to make a phone call is to wait for the phone to ring, lift the hook from the phone and start voice communication. Although this is a correct outcome, it may not be the result that the user expected, who had probably in mind to *initiate* a phonecall and not to wait for someone to call. This means that a more accurate formulation of the 'how to' question is necessary in order to obtain a more satisfactory answer. Actually, the user would like to know how to *initiate* a phonecall *without waiting for someone else to call*. This can be expressed by means of the until operator (see Appedix A.

[1] In SPIN process instances are assigned process identification numbers (pid) in the same style as in UNIX. Such numbers must be mentioned in the LTL formulas. The pid for our POTS process is 2.

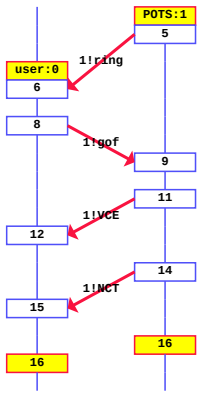

Fig. 2. Message Sequence Chart of shortest trace to make a phonecall

Let p be defined as before (`#define p (POTS[2]@SPEAKING)`) and let us define q as `#define q (POTS[2]@RINGING)`, then we would like to obtain a series of instructions that lead to p without q becoming true before p. So, formulated as an LTL formula: `(!q) U p`. Of course, in order to obtain a trace from SPIN, we need to negate this formula when using the verifier, so we obtain `!((!q) U p)`.

```
#define p (POTS[2]@SPEAKING)
#define q (POTS[2]@RINGING)
!(!(q) U p)
```

The resulting trace is visualised in the message sequence chart in Fig. 3:

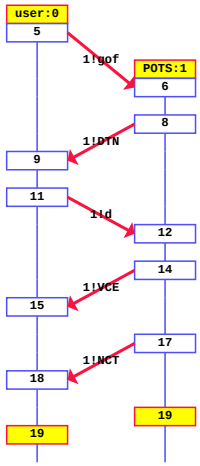

Fig. 3. How to make a phonecall without waiting for someone to ring

This trace gives indeed the set of instructions that answers the question of how to initiate a phone call, i.e. lift the hook, hear the dial tone, dial the number and start voice communication. Although the full trace generated by SPIN is rather detailed, it will be clear that the necessary information about the sequence of operations is easy to distil in an automatic way. A next step to produce a result that is resembling better the kind of text that can be found in user manuals is to relate each symbolic operation to a piece of text that tells the user in natural language what needs to be done [3]. In principle the traces may also be used as input to an animator that illustrates the user in a graphically animated way how to proceed.

In the following we give further examples of questions, their formalisations and resulting message sequence charts of the shortest series of instructions that can be given to instruct the user.

How to answer a ringing phone?
This question can be refined by requiring that the phone must be ringing, and in that case we are looking for the shortest series of instructions that lead to a situation that the user is speaking. Let p be defined as `#define p (POTS[2]@SPEAKING)` and q as `define q (POTS[2]@RINGING)`, this how-to question can then be formalised as `<>(q && <>p))`. The negation of this formula gives:

```
#define p (POTS[2]@SPEAKING)
#define q (POTS[2]@RINGING)
[] ! (q && <> p))
```

The resulting shortest trace is visualised in Fig. 4.

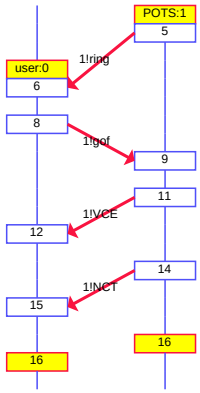

Fig. 4. How to answer a ringing phone?

What to do when the line is busy?
With a very similar pattern of LTL formulas we can answer questions of the type 'What if', like "What to do when the line is busy?". In this case the user is probably intending that (s)he would like to proceed with other activities, like making another phone call, when he discovers that the person (s)he was looking for is currently speaking to someone else. In this case, we can simply replace the definition of q by `#define q (POTS[2]@BUSY)` and re-use exactly the same formula as in the previous case:

```
#define p (POTS[2]@SPEAKING)
#define q (POTS[2]@BUSY)
[] ! (q && <> p))
```

We obtain the shortest trace of instructions shown in Fig. 5 (a):

If we want to express explicitly that the user wants to initiate another phone call instead of waiting for someone to ring, we need to be more precise. This is very similar to what we have seen in the second example:

```
#define p (POTS[2]@SPEAKING)
#define q (POTS[2]@BUSY)
#define r (POTS[2]@RINGING)
[] ! (q && ((!r) U p))
```

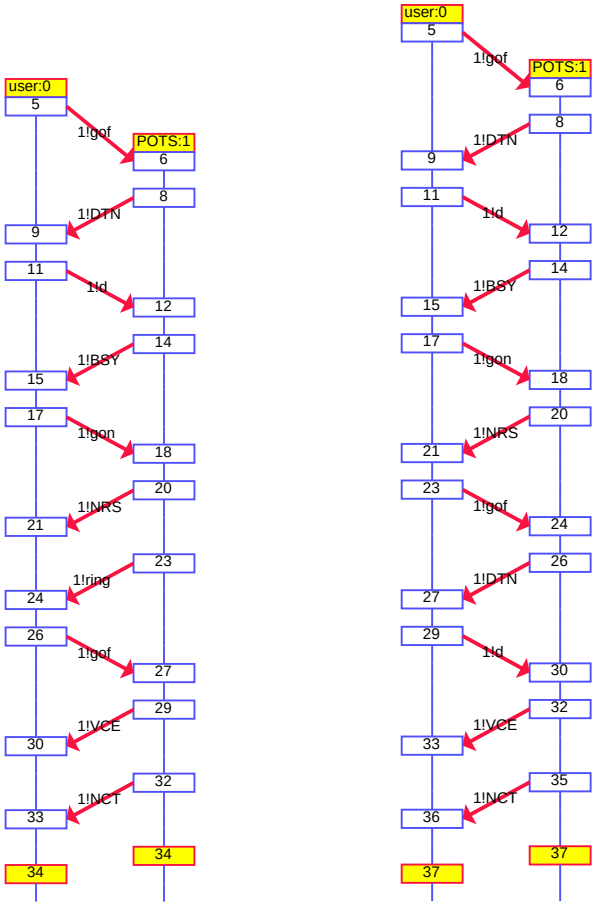

Fig. 5. (a) What to do when the phone gives busy? (b) What to do when the phone gives busy and I want to make another call?

Resulting in the series of instructions shown in Fig. 5 (b):

We could present many other examples, but the pattern of the LTL formulas would be essentially similar to the ones that we have presented. This means that the formulas do not need to be reinvented for every kind of question, thus facilitating their use. The concept of patterns or templates of temporal formulas has been proposed and is already used succesfully for the correctness analysis of formal specifications [7].

5 Conclusions

In this paper we have shown how the technique of model checking can be used to obtain automatically series of user instructions from formal specifications of a user interface. Such series of instructions form an important part of user manuals where the user is informed about the operation of a device or system. The automatic derivation of user instructions has several important advantages. First of all, it guarantees a close relation between the specification of the interface and the user manual and therefore it helps to avoid the introduction of errors in the documentation for the end-user. When a formal specification of the interface already exists for the purpose of correctness analysis, its use for the generation of instruction sets for a manual makes the specification even more useful. A third advantage is that the developer of the manual does not need to rely only on natural language descriptions of the system, which may be rather imprecise. The use of a model checker could be integrated into a special tool for the preparation of user manuals and safe the developer a lot of time. Finally, if a formal model of the interface is used also for the design of

the system or device, changes in the model can be easier translated into corresponding changes in the user manual.

In this preliminary paper we have only outlined the basic idea for the use of model checking for the generation of user instructions. Much can be done to improve the generated output. Instead of abstract symbols for operations and feedback readable text for a manual may be generated in successive steps [12].

A more exciting extension would be to explore whether the model checker could be used interactively in for example on-line helpers. This would require a translation of the on-line natural language question of the user into a proper logical formula and the immediate reply of instructions to the user. An even longer term project could be the use of a planner based on model checking such as MBP [1] that keep track of the current state of the system and the goal of the user, and that may be able to plan the right instructions even in cases where the user may have got disoriented. We leave this for future research for now.

6 Acknowledgements

The work in this report has been developed in the context of the C.N.R. coordinated project DimmiBene, CNRC00F31A.

References

1. P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a Model Based Planner. IRST Technical Report 0112-09, Istituto Trentino di Cultura, December 2001. In Proceedings of *IJCAI-2001: workshop on Planning under Uncertainty and Incomplete Information*, 2001.
2. E. M. Clarke, Jr., O. Grumberg, and D. Peled. Model Checking The MIT Press, Cambridge, Massachusetts, London, U.K., 1999.
3. B. De Carolis, S. Pizzutilo, F. de Rosis. User Manuals as Animated Agents. AI*IA Workshop on Agenti intelligenti e internet: teorie, strumenti e applicazionir, Milano, September 2000.
4. G. Doherty, M. Massink, and G. Faconti. Using hybrid automata to support human factors analysis in a critical system. *Formal Methods in System Design*, 19(2), September 2001.
5. E. A. Emerson. Temporal and Modal Logic. In: Handbook of Theoretical Computer Science, J. van Leeuwen (Ed.), Elsevier Science Publishers, 1990.
6. A. Fantechi, S. Gnesi, G. Ristori, M. Carenini, M. Vanocchi, and P. Moreschini. Assisting requirement formalization by means of natural language translation. Formal Methods in System Design, 4:243–263, 1994.
7. G. Holzmann. The model checker SPIN. IEEE Transactions on Software Engineering 23(5): pp. 279-295,1997.
8. D. Norman. The Psychology of Everyday Things. Basic Books, Inc, Publishers, New York, 1988.
9. A. Pnueli. The temporal logic of programs. Proceedings 18th IEEE Symposium on Foundations of Computer Science, pp. 46-57, 1977
10. Reiter, E., Mellish, C., and Levine, J. Automatic Generation of Technical Documentation. Applied Artificial Intelligence, Vol. 9, No. 3, pp. 259–287, 1995.
11. J. Rushby. Using model checking to help discover mode confusions and other automation surprises. In: Reliability Engineering and System Safety. Vol. 75, No. 2, pp. 167-177, 2002.
12. H. Thimbleby and P. Ladkin. From logic to manuals. In Proceedings of the IEEE Conference on Software Engineering, 1997.

A Linear Temporal Logic formulas

Temporal logic is a special type of Modal Logic and is used to describe and reason about how truth values of properties change over time. Pnueli was the first to recognise that such a logic could be a useful formalism for the specification and the verification of the correctness of computer programs. Typical operators provided by a system of Temporal Logic include *eventually* P which is true now if there is a future moment at which P becomes true and *always* P which is true now if P is true at all future moments. A third important operator is the *until* operator as in $P\ U\ Q$ which is true now if P is true and remains true at all future moments until Q becomes true. In a linear Temporal logic (LTL) temporal modalities are provided to describe events along a single time line. This is the kind of temporal logic used in the SPIN model checker and in this article.

In SPIN it is required to add a macro-definition for each propositional symbol used in the LTL formula. For example `#define p (a > b)` defines the proposition (p) that variable a is smaller than b. The propositional symbols can be used to compose linear temporal logic formulas by means of a number of temporal logic operators.

Valid temporal logic operators are:

`[]` Always
`<>` Eventually
U (Strong) Until
V The Dual of Until: $(pVq) == !(!pU!q)$

Some intuition for the linear-time operators is given in Fig. 6 where the black nodes indicate the moments where the proposition p is true, the grey nodes the ones where q is true. The associated temporal logic formulas are true in the respective initial node of the the time lines. All operators are left-associative (incl. U and V). The propositions used in the LTL formulas are buitl up from atomic propositions and Boolean operators.

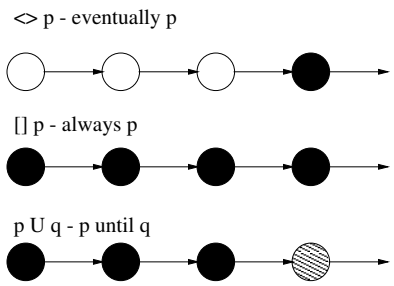

Fig. 6. Intuition for some linear-time operators [5]?

Boolean Operators:

`&&` Logical And (alternative form: / no spaces)
`!` Logical Negation
`||` Logical Or (alternative form: , no spaces)
`->` Logical Implication
`<->` Logical Equivalence

Boolean Predicates:

- `true, false`
- any name that starts with a lowercase letter

The are a number of frequently used LTL formulas for the verification of some important correctness properties. They are provided as templates by the SPIN tool.
Generic properties/Templates:

- Invariance: `[] p`
- Response: `p -> <> q`
- Precedence: `p -> (q U r)`
- Objective: `p -> <> (q || r)`

Each of the above 4 generic types of properties can (and will generally have to) be prefixed by temporal operators such as `[], <>, []<>, <>[]`. The last (objective) property can be read to mean that p is a trigger, or 'enabling' condition that determines when the requirement becomes applicable; then q can be the fullfillment of the requirement, and r could be a discharging condition that voids the applicability of the check.

For an overview and more formal introduction to LTL and other temporal logics we refer the interested reader to the chapter on Temporal and Modal Logic by E.A. Emerson [5].